Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Karin Frlic

Uporaba drevesnega preiskovanja Monte Carlo in strojnega učenja za učenje hevristične funkcije

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Aleksander Sadikov

Ljubljana, 2019

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Kandidatka naj razišče možnosti kombiniranja drevesnega preiskovanja Monte Carlo (MCTS) in strojnega učenja za pridobitev hevristične funkcije za uporabo pri klasičnem algoritmu minimaks. MCTS naj uporabi za ovrednotenje stanj v igri. Ta stanja naj opiše z množico atributov, ki jih pripravi vnaprej na podlagi svojega poznavanje igre in obstoječe literature. Na generiranih podatkih naj preizkusi različne algoritme strojnega učenja in s tem ustvari eno ali več hevrističnih funkcij. Kakovost igranja naj preveri na primeru igre Hex – igre s popolno informacijo za dva igralca.

Zahvaljujem se mentorju doc. dr. Aleksandru Sadikovu za velikodušno pomoč in podporo pri izdelavi diplomskega dela. Ves čas dela na projektu je z zanimanjem spremljal moj napredek in bil neusahljiv vir idej za izboljšave.

Zahvalila bi se rada tudi svoji družini. Najprej mami in atiju, da sta prenesla mojo neprestano odsotnost od doma, pa hkrati ves čas verjela vame in mi zaupala. Hvaležna sem tudi bratcu in sestricam, ki so me nenehno spraševali: „Kdaj boš končala igr'co?“ in mi s tem dvigali motivacijo za delo.

Najbolj iskrena zahvala pa gre Jakobu, ki je rad poslušal moje ideje in tarnanja ter včasih moje nerešljive težave rešil v trenutku. Znal mi je prisluhniti in je vedel, kdaj potrebujem spodbudo in kdaj opomin, da je čas za počitek.

Kazalo

Seznam uporabljenih kratic

kratica	**angleško**	**slovensko**
MCTS	Monte Carlo tree search	Drevesno preiskovanje Monte Carlo
UCT	Upper confidence bound applied to trees	Zgornja meja zaupanja pri drevesih
α-β	Alpha-beta pruning	Rezanje alfa-beta

Povzetek

Naslov: Uporaba drevesnega preiskovanja Monte Carlo in strojnega učenja za učenje hevristične funkcije

Avtor: Karin Frlic

Algoritem minimaks je eden najbolj razširjenih algoritmov za igranje iger med dvema igralcema. Pri tem se uporablja hevristična funkcija, ki ocenjuje, kako koristno je doseči neko stanje v igri za posameznega igralca. V diplomskem delu poskusimo tako funkcijo za igranje igre Hex ustvariti avtomatsko z uporabo različnih modelov nadzorovanega strojnega učenja. Učne primere za strojno učenje pridobimo s številnimi odigranimi igrami, ki jih simulira MCTS. Ugotovimo, da je igralec, ki za izbiro potez uporablja algoritem minimaks z α-β in naučeno funkcijo, slabši od igralca, ki igra samo z MCTS. Odkrijemo pa, da igralec, ki združi prednosti obeh omenjenih igralcev, igra bolje od MCTS.

Ključne besede: drevesno preiskovanje Monte Carlo, nadzorovano strojno učenje, algoritem minimaks, hevristična ocenjevalna funkcija, rezanje alfa-beta, igra Hex.

Abstract

Title: Using Monte Carlo tree search and machine learning to learn a heuristic function

Author: Karin Frlic

Minimax algorithm is one of the most widely used algorithms for playing two-player games. It uses a heuristic function that estimates the benefits of reaching a given game state for both players. In this bachelor thesis we attemp to automatically construct that kind of a function for the game of Hex. Different models of supervised machine learning are trained on learning samples, generated by simulations of MCTS. As a result, the player that uses minimax with α-β and the learnt function performs worse than the player that uses pure MCTS. However, the player combining advantages of both players achieves better results than MCTS.

Keywords: Monte Carlo tree search, supervised machine learning, minimax algorithm, heuristic evaluation function, alpha-beta pruning, the game of Hex.

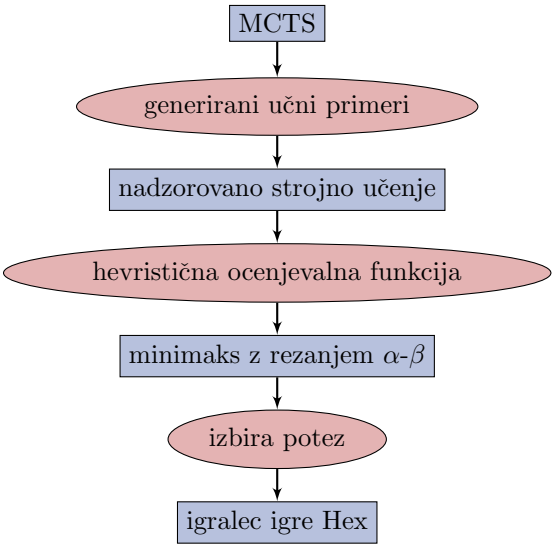
Poglavje 1

Uvod

Za raziskovalce umetne inteligence so namizne igre že dolgo zelo privlačno področje. Njihova jasno zastavljena pravila natančno definirajo vse možne poteze, zaradi svoje kompleksnosti pa zahtevajo učinkovite preiskovalne algoritme. Danes med najbolj uveljavljenimi algoritmičnimi pristopi k igranju najdemo MCTS in algoritem minimaks z α-β. Pri šahu, goju in marsikateri drugi igri je njihova uporaba pripeljala do močnih igralcev, ki se lahko kosajo s človeškimi prvaki.

Algoritem minimaks z α-β temelji na hevristični funkciji, ki ocenjuje, kako dobro je neko stanje v igri za posameznega igralca. Ponavadi to funkcijo izbere človek, ki zna določiti, katera stanja vodijo k zmagi in katera k porazu. A obstajajo igre, pri katerih to ali ni mogoče ali pa tega še ne znamo. Osrednji cilj diplomskega dela je raziskati možnosti avtomatskega generiranja ocenjevalne funkcije za uporabo pri takih igrah. Najprej uporabimo MCTS, ki generira učne primere za strojno učenje, tako da različna stanja v igri opiše z vrednostmi ročno izbranih atributov (vhodni podatki) in jim doda svojo oceno (izhodni podatek). Z odločitvenimi drevesi in linearno regresijo ustvarimo različne ocenjevalne funkcije in jih preizkusimo v algoritmu minimaks z α-β na primeru namizne igre Hex. Osnoven potek razvoja prikazuje slika 1.1.

V začetnem delu diplomskega dela spoznamo pravila in lastnosti igre Hex ter najpomembnejše že obstoječe programe, ki jo znajo igrati. Sledi poglavje,



Slika 1.1: Okvirna predstavitev sistema, ki ustvari igralca igre Hex. Razvit igralec poteze izbira z uporabo algoritma minimaks z α-β. Pri tem uporablja hevristično ocenjevalno funkcijo, ki je rezultat predhodnega nadzorovanega strojnega učenja. Učne primere za strojno učenje generira MCTS.

v katerem predstavimo uporabljene metode umetne inteligence: MCTS (podpoglavje 3.1), odločitveno drevo in linearno regresijo (podpoglavje 3.2) ter algoritem minimaks z α-β (podpoglavje 3.3). V poglavju 4 opišemo implementacijo vseh naštetih algoritmov, v 5. poglavju pa predstavimo rezultate dela. V zaključnem poglavju povzamemo ključne ugotovitve in predlagamo nadaljnje delo.

Poglavje 2

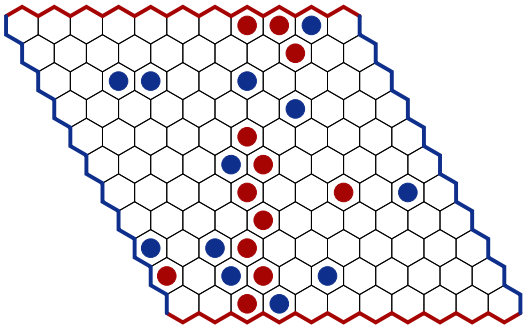
Igra Hex

V tem poglavju predstavimo igro Hex, na kateri temelji to diplomsko delo. Podpoglavje 2.1 razjasni pravila igre, v podpoglavju 2.2 so naštete nekatere njene lastnosti, doslej najuspešnejši že obstoječi igralci pa so opisani v podpoglavju 2.3.

2.1 Pravila igre

Pravila igre Hex so preprosta. Dva igralca na prosta polja igralne plošče izmenično postavljata kamenčke v svoji barvi. Velikost plošče, ki jo sestavljajo šestkotniki, postavljeni v obliki romba, ni standardizirana, največkrat pa se pojavlja v velikosti 11×11. Primer plošče je prikazan na sliki 2.1. Zmaga igralec, ki s svojimi kamenčki prvi vzpostavi neprekinjeno povezavo med svojim parom nasproti ležečih stranic plošče.

Igralec, ki začne, lahko vedno zmaga, če igra optimalno [5], zato se pogosto uporablja še dodatno pravilo: po začetni potezi prvega igralca se lahko drugi namesto za svojo potezo odloči za menjavo. V tem primeru s plošče odstrani postavljen kamenček prvega igralca, na zrcalno mesto pa postavi svojega. Nato je spet na vrsti prvi igralec. To pravilo močno zmanjša prednost prvega igralca, saj mora ta zdaj na začetku izbrati potezo, ki je za obe strani približno enako dobra.



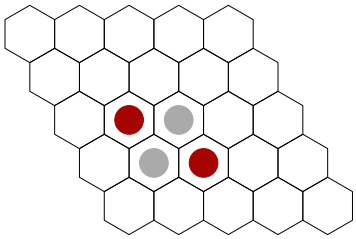
Slika 2.1: Primer plošče tekom igre. Rdeči igralec skuša povezati zgornjo in spodnjo stranico romba, modri pa levo in desno.

2.2 Lastnosti igre

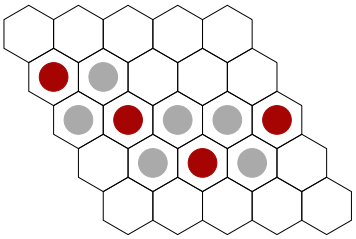
Igra Hex je deterministična (angl. *deterministic*) igra s popolno informacijo (angl. *perfect information*) in ničelnim izidom (angl. *zero-sum*).

Igra se ne more končati neodločeno, saj dokler igralec nima sklenjene celotne povezave, obstaja vsaj ena pot, ki vodi k zmagi nasprotnega igralca, če temu uspe zapolniti vsa prosta polja na tej poti s svojimi kamenčki. Kljub dokazani zmagovalni strategiji prvega igralca (brez dodatnega pravila menjave) [6] pa je težko najti zaporedje potez, ki vodijo do zmage. Eden izmed glavnih razlogov za to je velik vejitveni faktor, ki je na začetku igre enak številu polj na plošči (pri velikosti 11×11 torej 121), nato pa se z vsako potezo zmanjša za ena. Zaradi tega je izčrpno preiskovanje daljšega zaporedja potez močno oteženo. Natančneje, igra Hex na plošči poljubne velikosti $n \times n$ po zahtevnosti spada v razred PSPACE [12], kar pomeni, da je iskanje optimalne igralne strategije možno le na polinomskem prostoru. Še več, problem je PSPACE-poln, torej je enako težek kot najtežji problemi v tem razredu.

Pomembna struktura v igri so mostovi. Gre za dva kamenčka iste barve, med katerima sta dve prosti polji. Primer rdečega mostu prikazuje slika 2.2. Če nasprotnik (modri igralec) zasede eno izmed sivo označenih polj, lahko rdeči igralec obrani povezavo tako, da položi svoj kamenček na drugo sivo polje. Mostovi se lahko povezujejo v daljše virtualne povezave, kar je prikazano na sliki 2.3.



Slika 2.2: Most – rdeči igralec lahko označeni rdeči polji šteje za povezani.



Slika 2.3: Daljša virtualna povezava, sestavljena iz treh mostov.

2.3 Pregled obstoječih programov

Prvi programi za igranje te igre so za osnovo uporabljali algoritem minimaks z α-β, kasneje pa so prevladali programi osnovani na MCTS[1]. Spodaj opišemo tri pomembnejše programe: Hexy, njegovo izboljšavo Six in povsem drugačnega MoHex.

2.3.1 Prvi uspešni poskusi

Eden izmed najstarejših programov za Hex, ki se lahko kosa z najboljšimi človeškimi igralci, je Hexy iz leta 1999 [1]. Temelji na algoritmu minimaks z α-β. Pri tem uporablja ocenjevalno funkcijo, ki ploščo predstavi kot dve električni vezji, vsako za enega igralca. V teh vezjih ima vsako polje svoj upor: prazno polje vsebuje upor 1, polje z igralčevim kamenčkom ima upor 0, upor polja z nasprotnikovim kamenčkom pa je neskončen. Med vsaki sosednji polji plošče se doda povezava z uporom, ki je enak vsoti uporov obeh polj. Prav tako se doda povezava za vsako najdeno virtualno povezavo (program poleg povezav iz mostov išče tudi drugačne virtualne povezave, ki jih mi v tem delu ne omenjamo). Za vsakega igralca se nato izračuna skupni upor med njegovima stranicama plošče (R_1 za prvega in R_2 za drugega). Končna vrednost plošče je razmerje R_1/R_2. Manjši kot je rezultat, boljša je postavitev na plošči za prvega igralca in obratno. Opisana ocenjevalna

[1]Bralcu, ki katerega od omenjenih algoritmov ne pozna, predlagamo, da si najprej prebere poglavje 3.

funkcija je tako zanesljiva, da omogoča zelo uspešno igro kljub nizki globini, ki jo doseže preiskovanje igralnega drevesa.

Iz Hexy se je kasneje razvil Six, ki igra bolje zaradi dodanih hevrističnih omejitev, izboljšane ocenjevalne funkcije in zmanjšanega faktorja vejitve [2, 10]. Six je zmagoval na računalniški olimpijadi v igri Hex do leta 2006 [9], potem pa so ga začeli premagovati programi, ki temeljijo na drugačnih algoritmičnih pristopih.

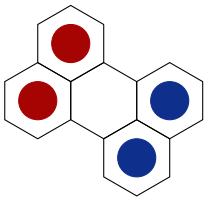
2.3.2 Danes najboljši program

Danes za najuspešnejši program velja MoHex, razvit na univerzi v Alberti leta 2007, ki za izbiro potez uporablja MCTS [2]. Algoritem najprej glede na vrednosti UCT v vozliščih drevesa izbere pot do enega izmed listov v drevesu, od tam naprej pa poteze izbira naključno. Edina situacija, kjer je poteza takoj določena, je primer, ko nasprotnik postavi svoj kamenček na eno izmed dveh polj igralčevega mostu. Takrat igralec takoj položi svoj kamenček na preostalo prosto polje.

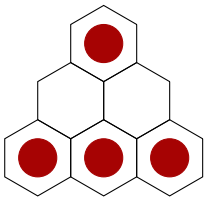
Da ni potrebno po vsaki potezi preverjati, ali ima kateri od igralcev sklenjeno povezavo med svojima stranicama, se za konec igre šteje trenutek, ko na igralni plošči ni več praznih polj. To ne more spremeniti izida igre, saj že obstoječe povezave ne more prekiniti noben dodatno položen kamenček, hkrati pa to pospeši preiskovanje drevesnih vozlišč.

Za zmanjšanje faktorja vejitve se uporablja analiza slabših polj (angl. *Inferior Cell Analysis*). Pri tem se na plošči poiščejo polja, za katera velja, da se vrednost stanja ne spremeni, če jih zapolnimo. Obstajata dve glavni vrsti takih polj: mrtva polja in zavzeta območja. Mrtva polja (primer na sliki 2.4) so tista polja na plošči, ki ne koristijo nobenemu igralcu. Zavzeta območja (primer na sliki 2.5) so skupine vsaj dveh povezanih celic, pri katerih lahko en igralec izniči vse koristi, ki jih ima drug igralec s postavitvijo svojega kamenčka v to območje. Vsa taka polja se zapolnijo s kamenčki ustrezne barve, kar zmanjša število prostih polj plošče in s tem število možnih potez.

Dodatno izboljšavo predstavljajo poskusi popolne analize postavitve na



Slika 2.4: Primer mrtvega polja. Na prosto polje lahko položimo rdeči ali moder kamenček, pa to ne bo koristilo nobenemu igralcu.



Slika 2.5: Primer zavzetega območja rdečega igralca. Modri igralec nima nobene koristi od polaganja svojega kamenčka na eno izmed prostih polj.

plošči. Ko je neko vozlišče dovoljkrat obiskano, skuša algoritem najti zmagovalno strategijo za enega izmed igralcev od tu naprej. Če jo najde, zavrže vse naslednike tega vozlišča in ob vseh kasnejših obiskih istega vozlišča kot rezultat vrne zmago igralca z zmagovalno strategijo.

Poglavje 3

Uporabljena metodologija

Za razumevanje diplomskega dela je potrebno poznavanje dveh metod umetne inteligence: preiskovalna drevesa in nadzorovano strojno učenje.

V prvem in zadnjem delu tega poglavja sta orisana dva algoritma, ki temeljita na uporabi dreves: MCTS (podpoglavje 3.1) in minimaks z α-β (podpoglavje 3.3). Drevesa, ki jih uporabljata, so sestavljena iz vozlišč, ki predstavljajo stanja v igri, in povezav med vozlišči, ki predstavljajo možne poteze v vsakem stanju. Prvi algoritem izbira poteze in gradi igralno drevo na podlagi velikega števila simuliranih iger, drugi pa s sistematičnim pregledom vseh možnih zaporedij potez določene dolžine.

Podpoglavje 3.2 predstavi osnove nadzorovanega učenja in dve njegovi metodi: odločitveno drevo, s katerim lahko predstavimo poljubno logično funkcijo, in linearno regresijo, ki lahko uspešno predstavi le podatke, pri katerih so izhodne vrednosti linearno odvisne od vhodnih.

3.1 Drevesno preiskovanje Monte Carlo

MCTS je metoda, ki koristnost posamezne poteze v igri določa na podlagi simulacij, s katerimi postopoma gradi in posodablja preiskovalno drevo. Vsako simulacijo lahko opišemo kot zaporedje štirih faz, ki so predstavljene v nadaljevanju tega poglavja: izbira vozlišča, razširitev vozlišča, simulacija iz

lista in vzvratno razširjanje. To zaporedje se ponovi poljubno mnogokrat, z naraščajočim številom ponovitev pa so ocene potez v igri vse bližje dejanskim (teoretičnim) vrednostim. Algoritem omejimo s številom iteracij, ki jih izvede, ali s časom, ki ga ima na voljo za izvajanje simulacij. Rezultat algoritma, tj. predvidoma najboljša možna akcija v dani situaciji, je poteza, ki vodi iz korena drevesa in je bila največkrat izvedena.

3.1.1 Potek iteracije MCTS

1. **Izbira vozlišča** (angl. *Selection*)

   V prvi fazi algoritma je izbran list preiskovalnega drevesa, iz katerega se bo simulacija nadaljevala. Začenši v korenu drevesa poteka izbira tako, da se za vsakega otroka o_i vozlišča s izračuna vrednost UCT po formuli:

   $$UCT(o_i) = \begin{cases} \infty & ; N(o_i) = 0 \\ \frac{S(o_i)}{N(o_i)} + C \times \sqrt{\frac{\ln(N(s))}{N(o_i)}} & ; sicer \end{cases},$$

   kjer je $N(v)$ število obiskov vozlišča v, $S(v)$ seštevek rezultatov, dobljenih z igro iz vozlišča v, C pa empirično določena konstanta. Prvi člen formule pripisuje višje vrednosti vozliščem, ki so se izkazala kot boljša od ostalih (težnja po izbiri potez, ki so se do sedaj izkazale za najboljše), drugi pa bolje oceni tista vozlišča, ki so bila manjkrat obiskana (želja ne spregledati dobre poteze). Razmerje med njima uravnava konstanta C.

   Izbran je otrok z najvišjo oceno UCT. Če je to list drevesa, se iz tega vozlišča nadaljuje naslednja stopnja simulacije, sicer se na tem vozlišču postopek izbire vozlišča rekurzivno ponovi.

2. **Razširitev vozlišča** (angl. *Expansion*)

   Drugi korak izbranemu listu drevesa doda otroke, po enega za vsako možno potezo iz stanja, ki ga to vozlišče predstavlja. Če drevo raste prehitro, lahko njegovo širitev upočasnimo s tem, da izbran list

razširimo le, če je bil obiskan že vsaj t-krat, pri čemer večji t pomeni počasnejše večanje drevesa.

Za nadaljevanje iteracije se naključno izbere enega izmed dodanih otrok.

3. **Simulacija iz lista** (angl. *Playout*)

   Od tu naprej algoritem naključno izbira poteze, dokler ne pride do stanja, v katerem je igra končana in rezultat znan.

   V nekaterih primerih je smiselno, da izbira potez v tej fazi ni povsem prepuščena naključju, temveč da vanjo vključimo nekaj domenskega znanja, ki pomaga ločevati med dobrimi in slabimi potezami. Na ta način je potek igre bolj podoben resnični igri, kar prispeva k hitrejšemu približevanju ocen dejanskim vrednostim, hkrati pa to upočasni simulacijo, kar vodi v manjše število iteracij v istem času [7].

4. **Vzvratno razširjanje** (angl. *Backpropagation*)

   V zadnjem delu simulacije se rezultat igre prenese do vrha drevesa. Ob tem se posodobijo vrednosti N in S vseh vozlišč na poti od lista, izbranega v fazi razširitve, do korena. Število obiskov N vozlišča v se poveča za 1, vrednosti S pa se prišteje rezultat igre za igralca, čigar poteza je vodila v v.

3.1.2 Prednosti in slabosti

MCTS je zelo razširjen algoritem, ker ga je enostavno uporabiti pri različnih igrah, saj za svoje delovanje potrebuje le pravila igre: začetno stanje, vse možne poteze iz nekega stanja skupaj z njegovimi nasledniki, v katere te poteze vodijo, in test, ki preverja, ali je igra že končana. To odpravi potrebo po iskanju hevristične funkcije, ki bi ocenjevala vrednost stanj, v katerih rezultat igre še ni znan. Poleg tega drevo raste neenakomerno – veje, ki se zdijo bolj obetavne, razvije bolj od tistih, za katere se zdi, da vodijo k porazu. Na ta način porabi manj časa v neperspektivnih vejah, kar je še posebej koristno pri igrah z velikim faktorjem vejitve.

Glavna slabost MCTS pa je povezana z naključnostjo algoritma. Ker se poteze v tretji fazi iteracije izbirajo naključno, se lahko zgodi, da se skoraj vsaka igra iz nekega vozlišča konča s porazom, čeprav bi se ob optimalni igri zaključila z zmago. Vrednost UCT za to vozlišče je zato nizka, zaradi česar ga MCTS obravnava kot manj perspektivnega. Enako se lahko zgodi v obratnem primeru, ko se večina naključno simuliranih iger konča z zmago, ker algoritem spregleda dobro nasprotnikovo potezo, zaradi katere bi se igra v resnici končala s porazom. To je še posebej vidno pri igrah kot je šah, kjer je neka poteza koristna le znotraj specifičnega zaporedja potez. Daljše kot je to zaporedje, dalj časa traja, da je odkrito, saj se MCTS sprva bolj osredotoča na ostale poteze, ki so dobre same po sebi.

3.2 Nadzorovano strojno učenje

Nadzorovano strojno učenje je pristop k strojnemu učenju, pri katerem algoritmu podamo učne primere v obliki parov vhodnih in izhodnih podatkov, njegova naloga pa je poiskati model, ki bo novim, prej nevidenim vhodnim podatkom določil izhodne vrednosti.

3.2.1 Vhod in izhod

Vhodni podatki vsakega učnega primera so predstavljeni kot seznam vrednosti atributov, ki so lahko različnih tipov [14]:

kategorični (angl. *nominal*) – vrstni red med vrednostmi ne obstaja, primerjati se jih da le po enakosti; primer: barva oči

vrstni (angl. *ordinal*) – vrednosti se lahko uredi po vrsti, a razlike med zaporednimi vrednostmi niso konstante; primer: ocene

intervalni (angl. *interval*) – razlike med vrednostmi so enakomerne, vrednosti se lahko seštevajo in odštevajo; primer: datum

razmernostni (angl. *ratio*) – vrednosti je smiselno tudi množiti ali deliti; primer: dolžina

Glede na izhod učenja ločimo dve veji, klasifikacijo in regresijo. Prvo uporabljamo za razvrščanje vhodnih podatkov v eno izmed končno mnogo vnaprej definiranih skupin, drugo pa za napovedovanje številčnih vrednosti na zveznem intervalu. V zaključku poglavja (3.2.3) sta opisana dva algoritma, med drugim uporabna za potrebe regresije: odločitveno drevo in linearna regresija.

3.2.2 Težave

Pogosta pojava pri modelih nadzorovanega učenja sta pretirano prilagajanje učnim podatkom (angl. *overfitting*) in nezmožnost opisa vseh značilnosti podatkov (angl. *underfitting*).

Pri pretiranem prilagajanju je problematično to, da model dobro predvidi izhodno vrednost pri podatkih, na katerih je bil naučen, dosti slabše rezultate pa dosega pri novih vhodnih podatkih. Do tega pride zaradi šuma v učnih podatkih, ki ga model prepozna kot eno izmed lastnosti podatkov in skladno s tem prilagodi svoj izhod. Če zaznamo, da naš model ne posplošuje dobro, je vredno razmisliti o njegovi poenostavitvi ali menjavi.

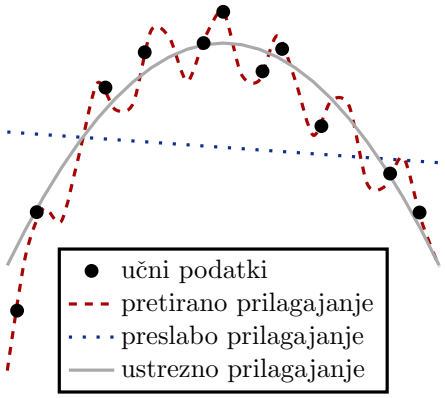
Če je napovedovanje nezadovoljivo že na učnih podatkih, pa je uporabljen model najverjetneje preveč enostaven in zato ne more pravilno upoštevati vseh atributov. Tu je rešitev v izbiri kompleksnejšega modela. Drugi možen razlog pa so slabo izbrani vhodni atributi, ki ne zagotavljajo dovolj informacij, potrebnih za točnejšo oceno.

Primera pretiranega in preslabega prilagajanja sta prikazana na sliki 3.1.

3.2.3 Primera

Odločitveno drevo

Čeprav se odločitveno drevo običajno uporablja za klasifikacijo, ga je možno uporabiti tudi kot regresijsko orodje. Tako drevo ima v vsakem notranjem



Slika 3.1: Primer pretiranega in preslabega prilagajanja učnim podatkom.

vozlišču test, ki preverja vrednost enega atributa. Za vsak možen rezultat tega testa vodi iz vozlišča ena povezava. V listih drevesa so številske vrednosti, ki predstavljajo možne izhode naučenega modela.

Pogost način izgradnje regresijskega drevesa je uporaba Huntovega algoritma, ki je sicer suboptimalen, a preprost in zadošča večini potreb [14]: Naj bo P_t množica vseh učnih podatkov, ki dosežejo vozlišče t. Glede na sestavo P_t so možna tri nadaljevanja:

1. Vsi učni primeri v P_t imajo enake vhodne podatke. V tem primeru postane t list drevesa, njegova vrednost pa je povprečje izhodnih podatkov vseh elementov v P_t.

2. Množica P_t je prazna. S tem postane t list drevesa z vrednostjo, ki je enaka povprečju izhodnih vrednosti v starševskem vozlišču.

3. Množica P_t vsebuje elemente z različnimi vhodnimi podatki. Izbere se atribut, na podlagi katerega se podatki v P_t razdelijo v vsaj dve podskupini. S tem postane t notranje vozlišče, algoritem pa se rekurzivno ponovi na vsaki podskupini.

Če ugotovimo, da se zgrajeno drevo preveč prilagaja učnim podatkom, lahko ta pojav omilimo na različne načine. Zahtevamo lahko, da sme algori-

tem vozlišče razdeliti na poddrevesa le, če to vozlišče vsebuje določeno najmanjše število učnih primerov ali če je po delitvi število elementov v vsaki podskupini nad določenim pragom. Za delitev se lahko odločamo tudi na podlagi variance izhodnih podatkov v vozlišču ali njenega zmanjšanja ob delitvi. Prav tako lahko določimo največjo globino drevesa ali največje število listov v drevesu.

Linearna regresija

Princip linearne regresije izhaja iz statistike. Njeno bistvo je iskanje koeficientov β, tako da se napovedan izhod $y_{napoved}(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n$ za vsak x čim bolj prilega podani izhodni vrednosti $y_{podan}(x)$. V formuli je x vhodni podatek, n število njegovih atributov, x_i pa njegov i-ti atribut.

Omenjeno prileganje se meri s povprečno vrednostjo kvadratov razlike med dejanskimi in napovedanimi izhodnimi vrednostmi za vse podane učne primere:

$$R = \frac{1}{|U|} \sum_{x \in U} (y_{podan}(x) - y_{napoved}(x))^2,$$

kjer je U množica vseh učnih primerov. Manjši R pomeni boljše ujemanje.

Učinkovitost linearnega modela je zelo odvisna od podatkov. Za dober rezultat morajo biti ti skladni z naslednjimi trditvami [3]:

1. Izhodni podatki so linearno odvisni od vhodnih.
2. Vrednosti posameznih atributov so normalno porazdeljene.
3. Posamezni atributi so med seboj neodvisni.
4. Posamezni podatki so med seboj neodvisni.
5. Napaka (R) je konstantna po celi regresijski osi.

3.3 Algoritem minimaks z rezanjem alfa-beta

V tem podpoglavju je predstavljen algoritem minimaks, primeren za iskanje igralne strategije v igri z ničelnim izidom, pri kateri dva igralca poteze izbirata izmenično.

3.3.1 Minimaks

Osnovna različica tega algoritma z iskanjem v globino preišče poddrevo s korenom v trenutnem stanju igre in podano globino. V vsakem listu tega poddrevesa izračuna predviden rezultat igre za igralca na potezi z uporabo vnaprej definirane ocenjevalne funkcije. Rezultat nato prenese navzgor po drevesu na sledeč način: v vsakem notranjem vozlišču poddrevesa se izbere skrajna vrednost iz njegovih otrok – pri vozlišču na lihi globini največja, pri vozlišču na sodi globini pa najmanjša vrednost[1]. S tem algoritem posnema realno igro, kjer igralec na potezi želi čim boljši rezultat zase, njegov nasprotnik pa mu želi čim bolj škodovati. Igralec, ki je na vrsti, na koncu izbere potezo, ki vodi iz korena drevesa do njegovega otroka z največjo vrednostjo.

Omenjena ocenjevalna funkcija je definirana tako, da vrne:

- pozitivno vrednost za stanja, ki so v prid igralcu na potezi,
- negativno vrednost za stanja, kjer ima prednost nasprotni igralec,
- 0 za stanja, ko imata oba igralca enake možnosti za zmago.

Bolj ko je ocena različna od 0, bolj izrazita je prednost enega izmed igralcev. Seveda pa je to le predvidena ocena, ki ne odraža nujno dejanskega stanja.

Različica algoritma, ki poenostavi programiranje, je negamaks [4] (prikazan v psevdokodi algoritma 1), ki upošteva dejstvo, da ima igra ničelni izid in je zato vrednost stanja za enega igralca negacija vrednosti za njegovega nasprotnika. Ob začetnem klicu algoritma kot argumenta podamo globino drevesa, ki naj bo pregledano, in trenutno stanje v igri.

[1] Privzamemo, da leži koren drevesa na globini 1.

Algoritem 1 Negamaks

```
procedure NEGAMAKS(globina, stanje)
    if KONČNO(stanje) or globina ≤ 0 then
        return OCENI(stanje)
    end if
    max ← −∞
    for all s iz OTROCI(stanje) do
        v ← − NEGAMAKS(globina − 1, s)
        max ← MAX(v, max)
    end for
    return max
end procedure
```

3.3.2 Rezanje alfa-beta

Najpogostejša optimizacija algoritma minimaks je dodajanje α-β, s katerim nekatere dele drevesa zavržemo, ne da bi jih pogledali. Ideja izhaja iz dejstva, da igralcu, ki že pozna vrednost ene poteze iz nekega stanja, ni potrebno v celoti pregledati neke druge veje drevesa iz istega stanja, za katero se je že izkazalo, da ne bo prinesla boljše poteze.

Algoritem za vsako drevesno vozlišče v na lihi višini hrani vrednost α, kamor je shranjena najvišja ocena, ki je bila do tega trenutka najdena med pregledovanjem možnih potez iz v. Za vsako vozlišče na sodi višini hrani vrednost β, ki shranjuje najnižjo tako oceno. Vrednost α v vozlišču s stanjem s torej predstavlja oceno poteze, ki jo bo izbral igralec na potezi v stanju s, sodeč po doslej pregledanih možnih potezah, β pa oceno poteze, ki jo bo izbral njegov nasprotnik. Iz tega je razvidno, da se α, ki ima začetno vrednost $-\infty$, v nekem vozlišču ne more nikoli zmanjšati, β z začetno vrednostjo ∞ pa ne povečati.

Ko je preiskovanje poddrevesa iz nekega vozlišča končano, se posodobi vrednost α oz. β njegovega predhodnika p. Če:

- p hrani α in je nova vrednost večja ali enaka vrednosti β katerega od

njegovih predhodnikov ALI

- p hrani β in je nova vrednost manjša ali enaka vrednosti α katerega od njegovih predhodnikov,

se zavrže vse še nepregledane veje iz p. Pokazalo se je namreč, da se igra z izbiro poteze, ki vodi v p, za igralca, ki jo izvede, zagotovo ne konča bolje kot v primeru izbire katere izmed prej pregledanih potez.

Sledeč zgledu minimaksa lahko tudi pri tej različici uporabimo ničelni izid igre, saj vemo, da je vrednost najbolje ocenjene poteze enega igralca enaka negaciji najslabše ocene nasprotnikove poteze [4]. Tako strukturo prikazuje psevdokoda algoritma 2. Poleg prvih dveh argumentov, ki sta enaka kot pri negamaksu, pri klicu algoritma podamo še vrednosti $\alpha = -\infty$ in $\beta = \infty$.

Algoritem 2 Negamaks z α-β

```
procedure NegamaksAB(globina, stanje, α, β)
    if Končno(stanje) or globina ≤ 0 then
        return Oceni(stanje)
    end if
    max ← −∞
    for all s iz Otroci(stanje) do
        v ← − NegamaksAB(globina − 1, s, −β, −α)
        max ← Max(v, max)
        if max > α then
            α ← max
            if α ≥ β then
                break
            end if
        end if
    end for
    return max
end procedure
```

Doprinos α-β k minimaksu

Minimaks z α-β vrne enak rezultat, kot bi ga osnoven minimaks, le v krajšem času, saj v veliki večini primerov pregleda manjši del preiskovalnega drevesa. Če je b vejitveni faktor drevesa in d globina, ki jo mora algoritem preiskati, potem minimaks vedno pregleda vseh b^d vozlišč, medtem ko je učinkovitost α-β močno odvisna od vrstnega reda pregledovanja potez. Ob najugodnejši razporeditvi, ko so možne poteze iz enega stanja že na začetku razporejene od najboljše do najslabše, gre minimaks z α-β le skozi $b^{d/2}$ vozlišč. Če so poteze razporejene naključno, to število naraste na $b^{3d/4}$, v najmanj ugodnem primeru, ko so poteze v vsakem vozlišču pregledane od najslabše do najboljše, pa α-β ne more zavreči nobene veje drevesa in tako kot minimaks pregleda vsa vozlišča [13].

3.3.3 Sortiranje potez

Da α-β omogočimo čim boljše rezanje drevesa, je torej smiselno urediti poteze glede na njihovo oceno, preden jih raziščemo. Ker natančnih vrednosti ne poznamo (če bi jih, iskanje sploh ne bi bilo potrebno), potrebujemo način, kako jih čim bolje oceniti. Za to lahko uporabimo lastnosti poteze ali stanja, v katerega poteza vodi, na primer tip poteze (postavitev figure na sredinsko ali robno polje, premik naprej ali nazaj) ali število in vrsto figur na plošči. Drug način je uporaba iterativnega poglabljanja, s katerim postopoma večamo globino iskanja – najprej preiščemo drevo globine 1, nato 2, zatem 3 in tako naprej do izbrane največje globine. Za urejanje potez v drevesu globine $d+1$ uporabimo ocene potez, ki smo jih dobili v preiskovanju drevesa globine d.

3.3.4 Transpozicijska tabela

Še ena možna dodelava algoritma minimaks, posebej primerna za igre, kjer je enako stanje v igri mogoče doseči z več različnimi zaporedji potez, je dodajanje transpozicijske tabele. Ko je neko vozlišče do konca preiskano, se v tabelo shrani njegova vrednost skupaj s stanjem, ki ga vozlišče vsebuje,

in globino poddrevesa, na podlagi katerega je bila ta ocena pridobljena. Če je ob naslednjem obisku vozlišča z istim stanjem shranjena globina večja ali enaka globini poddrevesa, ki bi bilo pregledano ob tej iteraciji, se namesto ponovnega preiskovanja uporabi shranjena vrednost. V nasprotnem primeru se poddrevo preišče in stari zapis v transpozicijski tabeli prepiše.

Transpozicijsko tabelo lahko koristimo tudi pri sortiranju potez, saj shranjene vrednosti stanj predstavljajo dobro osnovo za ločevanje dobrih in slabih potez. Vsakič, ko želimo pregledati naslednike nekega stanja, najprej preverimo, ali so ta stanja že v transpozicijski tabeli. Tista, ki so, uredimo po shranjeni vrednosti in jih pregledamo od najbolje do najslabše ocenjenega. Stanja, ki jih v transpozicijski tabeli še ni, v pregledovanje vključimo na poljubnem mestu.

Poglavje 4

Implementacija

V tem poglavju predstavimo, kako smo povezali algoritme, opisane v 3. poglavju, in ustvarili igralca igre Hex. V začetku poglavja opišemo implementacijo igre Hex, ki smo jo uporabili pri MCTS in algoritmu minimaks. Sledi podpoglavje o MCTS, s katerim smo pridobili učne podatke za nadzorovano strojno učenje, čigar implementacija je opisana v podpoglavju 4.3. Rezultat učenja sta bili hevristični funkciji, ki smo ju (vsako zase) uporabili v algoritmu minimaks z α-β za igranje igre Hex. Implementacija tega je predstavljena v podpoglavju 4.4. Poglavje zaključimo s predstavitvijo razvitih igralcev in opisom izdelanega grafičnega uporabniškega vmesnika.

Igra Hex, MCTS, minimaks z α-β in strežniški del grafičnega vmesnika so napisani v programskem jeziku Go, za strojno učenje smo uporabili Python, za prikaz uporabniškega vmesnika v internetnem brskalniku pa smo potrebovali še JavaScript, HTML (*Hyper Text Markup Language*, sl. *jezik za označevanje nadbesedila*) in CSS (*Cascading Style Sheets*, sl. *kaskadne stilske podloge*).

Pripadajoča izvorna koda je pod pogoji licence *BSD 2-Clause License* na voljo na spletnem naslovu github.com/RdecKa/0xAI .

4.1 Igra Hex

Za globlje razumevanje preostanka diplomskega dela tu opišemo, kako smo predstavili stanja v igri in kako smo preverjali, kdaj je igra končana. Na koncu naštejemo atribute, ki jih lahko pripišemo postavitvi kamenčkov na plošči.

4.1.1 Predstavitev stanja v igri

Da bi bilo upravljanje s stanji v igri čim hitrejše, smo se odločili za uporabo bitnih igralnih plošč. V ta namen vsako stanje v programu predstavimo kot seznam celih števil. Vsako število predstavlja eno vrsto polj na plošči, torej je dolžina seznama enaka številu vrstic na plošči, v našem primeru je to 11[1]. Po dva zaporedna bita števila predstavljata zasedenost neke celice na način, ki je prikazan v tabeli 4.1, do vrednosti pa dostopamo s pomočjo bitnih operacij, predvsem operacije IN (angl. *AND*) in zamikov (angl. *bit shifts*). Najnižja dva bita vsebujeta informacijo o najbolj levi celici v vrsti. Ker smo uporabili 32-biten zapis nepredznačenih celih števil, lahko predstavimo plošče, katerih širina je največ 16. Določili smo, da rdeči igralec povezuje zgornjo in spodnjo vrsto polj na plošči, modri pa levi in desni rob.

Zaporedna bita	**Pomen**
00	Prazno polje
01	Rdeče polje
10	Modro polje
11	Nedefinirana vrednost

Tabela 4.1: Pomen bitov v predstavitvi postavitve kamenčkov na plošči

[1]Orientiranost plošče v igri ni pomembna, a smo privzeli, da so vrste vodoravne, vsaka vrsta pa je glede na svojo predhodnico zamaknjena za pol polja v desno.

Primer zapisa

Podajmo zapis stanja, ki ga prikazuje slika 2.1:

$$[606208, 65536, 8352, 32768, 1024, 1536, 540928, 256, 98, 2145, 144].$$

Za primer razložimo spodnjo vrsto plošče, ki je zapisana kot 144. V tabeli 4.2 je to število zapisano v dvojiški obliki, vsak par zaporednih bitov pa je označen z barvo kamenčka, ki jo ta par predstavlja. Najnižja bita predstavljata najbolj levo celico v vrsti, zato je zapis v tabeli zrcalen sliki plošče.

Dvojiško	00	00	00	00	00	00	00	00	00	00	00	00	10	01	00	00
Polje	/	/	/	/	/	.	.	.	.	.	.	.	M	R	.	.

Tabela 4.2: Primer zapisa ene vrstice igralne plošče z 32 biti. M pomeni moder kamenček, R rdeč kamenček, pika prazno polje, poševnica pa neuporabljene bite.

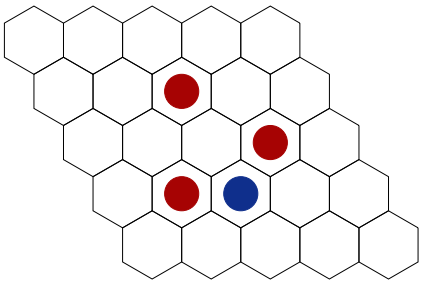
4.1.2 Preverjanje končnega pogoja

Igra je končana, ko ima eden izmed igralcev sklenjeno neprekinjeno povezavo med svojima robovoma plošče. Da bi se igra iztekla hitreje, smo se odločili, da igro končamo že v trenutku, ko ima igralec virtualno povezavo med svojima stranicama, saj bo ta igralec zagotovo zmagal, razen če to povezavo spregleda.

Za odkrivanje povezav med stranicami smo uporabili iskanje A*. Algoritem začne z iskanjem na eni izmed stranic plošče in pregleduje polja ustrezne barve, dosegljiva preko neposredne ali virtualne povezave. Iskanje vodi hevristična funkcija, ki računa oddaljenost doseženega polja od nasprotne stranice. Če je med iskanjem doseženo polje na nasprotni strani plošče, je bila povezava najdena in se igra lahko šteje za končano. Dovolj je, če se postopek izvede samo za igralca, ki je zadnji izvedel potezo.

Pri vključevanju virtualnih povezav je bilo potrebno upoštevati eno izjemo, ko dva zaporedna mostova ne tvorita virtualne povezave. Primer take

situacije prikazuje slika 4.1. Če modri igralec postavi svoj kamenček na sredinsko polje, rdeči ne more obraniti obeh svojih mostov, saj lahko v eni povezavi na ploščo postavi le en kamenček.



Slika 4.1: Primer, ko dva povezana mostova ne tvorita virtualne povezave.

4.1.3 Izbrani atributi

Za kasnejše strojno učenje je bilo potrebno vsako stanje opisati z več atributi. Lastnosti plošče, ki smo jih opazovali, so bile:

- zadnji igralec na potezi,
- število vseh kamenčkov na plošči,
- število vrstic z vsaj enim rdečim kamenčkom (enako za modre),
- število stolpcev z vsaj enim rdečim kamenčkom (enako za modre),
- vsota razdalj vseh rdečih kamenčkov do sredine plošče (enako za modre),
- število praznih polj plošče, ki se neposredno dotikajo polj z rdečimi kamenčki ali so z njimi virtualno povezani (enako za modre),
- število 24 različnih vzorcev kamenčkov na plošči (ločeno za rdeče in modre kamenčke), naštetih v dodatku A.

4.2 Drevesno preiskovanje Monte Carlo

V tem podpoglavju opišemo našo implementacijo MCTS in postopek generiranja učnih podatkov za kasnejšo uporabo v nadzorovanem strojnem učenju.

4.2.1 Potek iteracije

Pri programiranju algoritma smo natančno sledili zapisanemu v poglavju 3.1, tu naštevamo le posebnosti. Pri računanju vrednosti UCT smo konstanti C določili vrednost $\sqrt{2}$. V fazi razširitve vozlišča smo izbran list drevesa razširili le, če je bil prej obiskan že vsaj desetkrat, s čimer smo upočasnili sicer zelo hitro rast drevesa. Med simulacijo iz lista smo poteze izbirali povsem naključno vse dokler nismo prišli do stanja, v katerem je imel eden izmed igralcev (virtualno) povezavo med svojima stranicama. Iz tega stanja smo poslali rezultat igre (ta je vedno 1, saj vedno zmaga igralec, ki naredi zadnjo potezo) navzgor po drevesu, ob tem pa smo ga na vsakem nivoju negirali in prišteli vrednosti S vsakega vozlišča na poti.

4.2.2 Pridobivanje učnih primerov

MCTS smo uporabili za generiranje parov vhodnih in izhodnih podatkov za strojno učenje. V ta namen smo iskanje pognali iz različnih igralnih stanj, vsakič s časovno omejitvijo 10 sekund. Po preteku tega časa smo vsako vozlišče drevesa v, ki je bilo obiskano vsaj 1000-krat, shranili kot učni primer: v datoteko smo zapisali vrednosti vseh atributov stanja v vozlišču v, naštetih v poglavju 4.1.3 (vhodni podatki), in rezultat $\frac{S(v)}{N(v)}$ (izhodni podatek). Otroke izpisanih vozlišč, ki sami niso dosegli praga za izpis, smo dodali v seznam možnih novih začetkov programa.

Za nadaljevanje MCTS se je najprej izbralo število kamenčkov n, za katerega je bil produkt n in števila predhodnih začetkov iskanja iz stanj z n kamenčki najmanjši. Nato se je naključno izbral eden izmed vseh možnih kandidatov v seznamu z n kamenčki na plošči. Na ta način smo dosegli, da

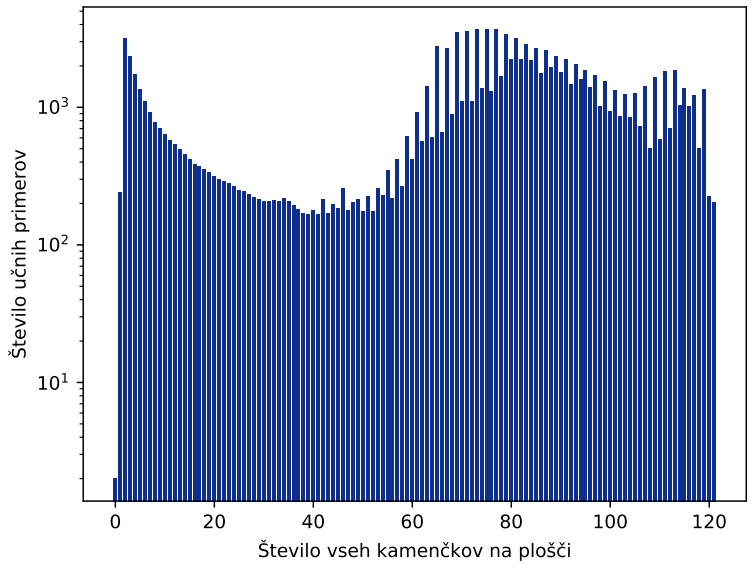
MCTS razišče plošče vseh zasedenosti, hkrati pa daje prednost ploščam z začetka igre, saj je dober začetek zelo pomemben za uspešno igro.

Prvo iskanje smo začeli v začetnem stanju igre, ko na plošči ni še nobenega kamenčka, v nadaljevanju pa smo za pohitritev generiranja učnih primerov izvajali več (6) iskanj hkrati. Možnost, da se isto stanje pojavi v različnih iskanjih, smo zaradi velike širine drevesa zanemarili.

4.3 Nadzorovano strojno učenje

Ustvarjene učne podatke smo analizirali in preverili njihovo raznolikost. Ob tem smo si pri risanju grafov v Pythonu pomagali z dvema programskima knjižnicama: *Matplotlib* [8] in *Seaborn* [15]. Izdelali smo dva modela, ki sta se iz teh podatkov naučila stanjem v igri pripisovati njihove vrednosti glede na koristnost za posameznega igralca.

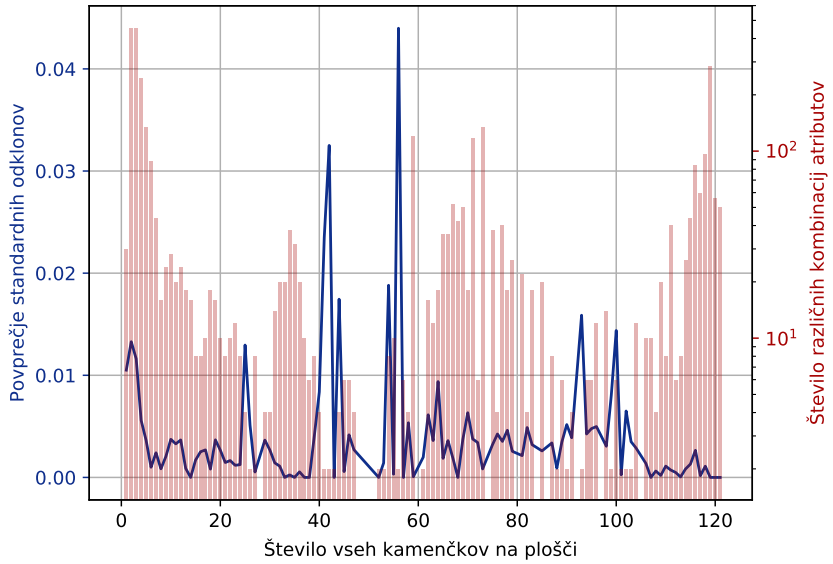
4.3.1 Analiza podatkov



Slika 4.2: Razporeditev učnih podatkov glede na zasedenost igralne plošče.

Pridobili smo 128.714 učnih podatkov. Njihova razporeditev glede na število vseh kamenčkov na plošči je prikazana na grafu 4.2. Graf kaže, da so zastopane vse možne pokritosti plošče, več je primerov z začetka in konca igre.

Da bi preverili kvaliteto podatkov, smo združili učne primere, ki so imeli povsem enake vrednosti atributov. Dobili smo več skupinic, izmed katerih smo odstranili vse, ki so vsebovale le en element. Za vsako izmed preostalih skupin smo izračunali povprečje in standardni odklon izhodnih podatkov, nato smo jih združili v večje skupine glede na število kamenčkov. Izračunali smo povprečje standardnih odklonov v vsaki izmed večjih skupin in jih prikazali na grafu 4.3.



Slika 4.3: Moder graf prikazuje povprečje standardnih odklonov znotraj skupin z enakim atributnim zapisom, združenih glede na število kamenčkov. Rdeč graf kaže število teh skupin.

Odstopanja ocen so majhna, redko nad 0,01, torej so stanja z enakimi atributi večinoma ocenjena podobno. Sodeč po tem sklepamo, da so atributi, s katerimi opisujemo postavitve na ploščah, izbrani dovolj dobro, da je na podlagi njihovih vrednosti mogoče dobro oceniti vrednost stanja.

4.3.2 Posebnosti pri posameznem modelu

Za učenje modelov smo uporabili knjižnico *scikit-learn* [11]. Posebno pozornost smo namenili temu, da smo modelu omogočili prilagoditev ocenjevanja glede na polnost plošče in barvo igralca, ki je na potezi.

Odločitveno drevo

Prvi model smo naučili s pomočjo *DecisionTreeRegressor* iz *scikit-learn*. Za preprečitev pretiranega prilagajanja učnim podatkom smo poskušali z omejitvijo drevesne globine in nastavljanjem najmanjšega dovoljenega števila elementov v listih drevesa, a se je izkazalo, da to nima velikega vpliva na rezultat. Ko smo 90% učnih primerov uporabili za učenje, 10% pa za testiranje, je bila uspešnost modelov zelo podobna, kar kaže tabela 4.3. Uspešnost smo merili z uporabo funkcije *DecisionTreeRegressor.score*, ki za izračun uporablja formulo

$$S = 1 - \frac{\sum_{x \in T}(y_{podan}(x) - y_{napoved}(x))^2}{\sum_{x \in T}(y_{podan}(x) - \frac{1}{|T|}\sum_{x' \in T} y_{podan}(x'))^2}.$$

V formuli T predstavlja množico testnih primerov. Dobljena ocena je lahko največ 1, navzdol pa ni omejena in je lahko negativna.

Omejitev globine	**Najmanjše število v listu**	**Uspešnost (S)**
5	5	0,970
10	5	0,988
Brez omejitve	1 (Brez omejitve)	0,994
Brez omejitve	10	0,990
Brez omejitve	20	0,989
Brez omejitve	50	0,986

Tabela 4.3: Uspešnost modela glede na nastavljene parametre.

Ko smo kasneje primerjali igralce med sabo, smo uporabili drevo brez omejene globine in brez omejevanja števila elementov v listih. Ob tem smo

ugotovili, da igralec, ki za izbiro potez uporablja ta model, kljub izredni točnosti ocenjevanja ne igra preveč dobro. Preverili smo, kako uspešen je model pri ocenjevanju plošč različnih polnosti, in opazili, da je model zanesljiv pri zelo praznih (do 4 kamenčki) ter zelo polnih ploščah (nad 60 kamenčkov), kjer je njegova točnost ocenjena z več kot 0,85, pri ploščah s sredine igre pa dosega slabše rezultate, okoli 0,5. Zelo dobra splošna točnost je posledica tega, da je velika večina testnih primerov predstavljala stanja z več kot 60 kamenčki. Na enak način smo preverili še uspešnosti modelov z drugačnimi vrednostmi parametrov in opazili enak trend. Edina izjema je bil prvi model z globino, omejeno na največ 5 nivojev. Njegova točnost pri napovedovanju vrednosti plošč z do 10 kamenčki je bila ocenjena negativno.

Odločili smo se obdržati prvotno izbran model. Analizirali smo ga in poiskali atribute, ki najbolj vplivajo na ocenjevanje. Z *DecisionTreeRegressor.feature_importances_* smo za vsak atribut pridobili informacijo o tem, koliko je ta atribut prispeval k zmanjšanju variance izhodnih podatkov ob delitvi vozlišč drevesa na manjše skupine. Rezultati so pokazali, da so najbolj vplivni atributi naslednji vzorci (predstavljeni v dodatku A): red_p9 (0,472), blue_p14 (0,225) in red_p17 (0,142). Izmed preostalih atributov so le še štirje dobili oceno, višjo od 0,01: red_p15 (0,058), blue_p3 (0,035), blue_p9 (0,020) in blue_p19 (0,017). Ostali atributi so dobili manjše vrednosti, a vsi več od 0, torej ima vsak od njih vsaj kdaj vpliv na ocenjevanje.

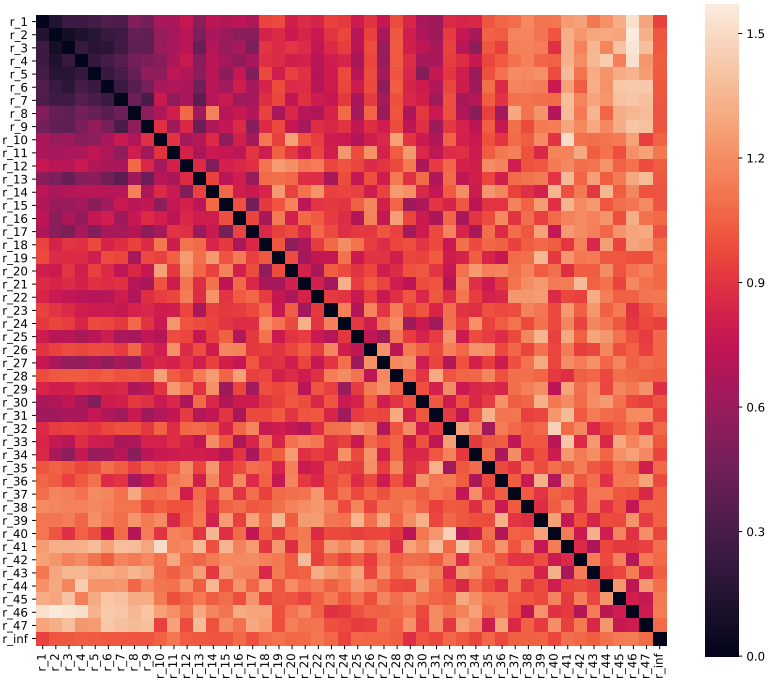
Dodatno delo, da bi modelu omogočili različno ocenjevanje glede na zasedenost plošče ali barvo igralca, ni bilo potrebno, saj lahko odločitveno drevo samo poskrbi za to z delitvijo učnih primerov v več skupin.

Linearna regresija

Pri uporabi linearne regresije je bilo potrebnega nekaj več priprav, saj model sam po sebi ne more zadovoljivo ocenjevati vseh možnih postavitev na plošči. Učne podatke smo najprej razdelili na dva dela glede na igralca, ki je izvedel zadnjo potezo v igri. Znotraj teh dveh delov smo učne primere združili v več skupin, tako da so imeli elementi znotraj posamezne skupine podobno

zasedenost plošče. Model smo nato sestavili iz več komponent, tako da smo na vsaki skupini uporabili *LinearRegression* iz *scikit-learn*. S tem smo dobili različne ocenjevalne funkcije, izmed katerih vsaka pokriva svoj del možnih stanj v igri.

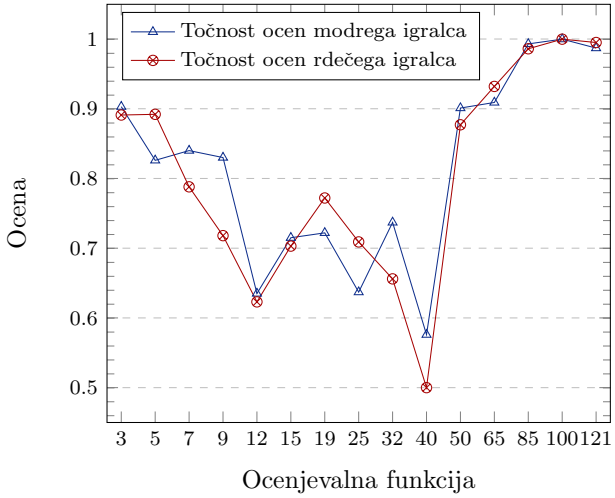
Za določitev meja med zasedenostmi plošče, na podlagi katerih smo razdelili učne primere, smo najprej poskusili deliti po vsakem položenem kamenčku. Za tem smo izračunali kosinusne razdalje med vsemi vektorji koeficientov linearne regresije (β) in jih (zaradi boljše preglednosti le do skupine s 47 kamenčki) prikazali na toplotnem zemljevidu 4.4. Prikazan je graf koeficientov, pridobljenih v skupinah, kjer je zadnjo potezo odigral rdeč igralec. Graf za modrega igralca je zelo podoben.



Slika 4.4: Vizualizacija kosinusnih razdalj med vektorji koeficientov linearne regresije. Vsaka vrstica in vsak stolpec predstavljata en vektor, številke v oznakah pa število kamenčkov, pri katerem smo razdelili učne primere. Temnejša barva/nižja vrednost pomeni večjo podobnost med dvema vektorjema, razdalja med dvema povsem enakima vektorjema je 0.

Opaziti je mogoče, da so si koeficienti na začetku igre zelo blizu, zato smo jih združili. Prav tako smo združevali učne primere z bolj zapolnjenimi ploščami. S tem smo želeli zmanjšati kompleksnost celotnega modela in priskrbeti več podatkov za učenje posameznim komponentam. Na koncu smo meje določili pri naslednjih številih kamenčkov: 3, 5, 7, 9, 12, 15, 19, 25, 32, 40, 50, 65, 85 in 100. Ker smo učne primere že na začetku ločili po barvi igralca, ki je izvedel zadnjo potezo, smo tako dobili 30 različnih ocenjevalnih funkcij.

Dobljene funkcije smo testirali na testnih podatkih in njihove točnosti izračunali z *LinearRegression.score*, ki uporablja isto funkcijo kot *DecisionTreeRegressor.score*. Rezultate prikazuje graf 4.5.



Slika 4.5: Točnost ocenjevalnih funkcij. Vsaka funkcija je bila naučena in testirana na svojem delu učnih primerov, izbranem glede na število kamenčkov na plošči. Oznaka na osi x predstavlja zgornjo mejo zasedenosti plošče, spodnja meja pa je za 1 večja od oznake leve sosede. Na primer, peta funkcija z leve ima zgornjo mejo 12, spodnjo pa 10. Prva funkcija ima spodnjo mejo 0.

Očitno je, da tako kot pri odločitvenem drevesu uspešnost skozi igro pada. Menimo, da je to posledica MCTS, ki večini stanj z začetka igre pripisuje vrednosti blizu 0, pri stanjih z več kamenčki pa so vrednosti bolj razpršene

med -1 in 1 ter zato težje ocenljive[2]. Izjemno natančne ocene stanj na zelo polnih ploščah (nad 65 kamenčkov) pojasnjujemo s tem, da je pri taki dolžini igre rezultat že precej odločen, zato je taka stanja lažje oceniti.

Preverili smo še, katere atribute uporablja posamezna ocenjevalna funkcija za določanje vrednosti stanjem. Izkazalo se je, da večina funkcij uporablja vse atribute, ki so smiselni pri neki polnosti plošče. Primera nesmiselnih atributov sta vzorec, sestavljen iz več kamenčkov, kot jih je na plošči, in število zasedenih vrstic ali stolpcev pri skoraj povsem polni plošči.

4.3.3 Izpis kode

Da bi naučeni ocenjevalni funkciji čim lažje uporabili v algoritmu minimaks, ki je napisan v programskem jeziku Go, smo modela izpisali v obliki kode v tem jeziku. Del tako pridobljene kode je prikazan v dodatku B.

4.4 Algoritem minimaks z rezanjem alfa-beta

Naučeni ocenjevalni funkciji smo uporabili za predvidevanje vrednosti stanj v algoritmu minimaks z α-β. V tem podpoglavju naštejemo posebnosti pri implementaciji tega algoritma.

4.4.1 Postopek izbire potez

Za izbiro potez smo implementirali negamaks po zgledu psevdokode algoritma 2. Uporabili smo iterativno poglabljanje, pri čemer smo postopoma povečevali globino pregledanega drevesa. Preiskovanje smo začeli z največjo dovoljeno globino 2, vsaka naslednja iteracija pa je pregledala dva nivoja drevesa več od prejšnje. Na ta način smo vedno ocenjevali stanja na sodih globinah drevesa, s čimer smo pri vsaki potezi igralca, ki je bil na vrsti, upoštevali tudi nasprotnikov odgovor.

[2]Ocena 0 pomeni, da imata oba igralca enakovredne možnosti za zmago, 1 pomeni zmago rdečega, -1 pa zmago modrega igralca.

Iskanje smo časovno omejili, po izteku časa pa smo ga prekinili in vrnili potezo, izbrano pri zadnji iteraciji negamaksa, ki se je v celoti zaključila. Zagotovili smo dovolj časa, da je program zagotovo lahko končal vsaj preiskovanje drevesa do globine 2.

4.4.2 Optimizacija

Zaradi velikega vejitvenega faktorja igre Hex je naš program zelo počasi napredoval v globino. Odločili smo se, da poskusimo iskanje pohitriti z uporabo transpozicijske tabele in sortiranja potez. Ker je Hex igra, pri kateri je mogoče isto stanje doseči na veliko različnih načinov, smo pričakovali, da se bo predvsem prva izboljšava izkazala za učinkovito. V koristnost sortiranja potez smo zaradi majhne dosežene globine drevesa sprva dvomili.

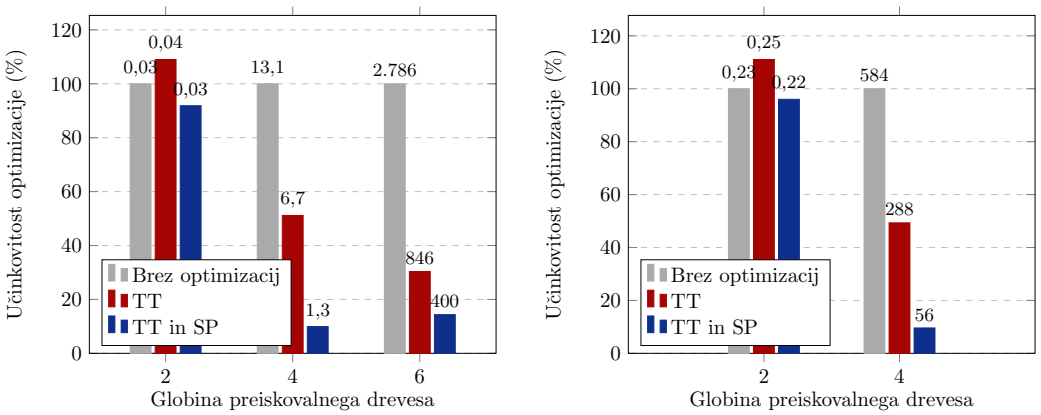
Najprej smo dodali transpozicijsko tabelo, kamor smo shranjevali pregledana stanja in njihove vrednosti. Ker lahko v igri Hex do iste postavitve kamenčkov pride po veliko različnih poteh, a vedno na isti globini igralnega drevesa[3], smo transpozicijsko tabelo poenostavili. Ugotovili smo, da bi vsaka nova iteracija iterativnega poglabljanja prepisala shranjene vrednosti, saj bi iskala globlje, vsak nižji nivo drevesa pa prinaša nova stanja, ki pred tem še niso bila videna. Zaradi tega v tabeli nismo hranili informacije o globini, temveč smo za vsako iteracijo naredili novo tabelo.

Dodali smo tudi sortiranje potez na podlagi ocen vrednosti stanj, v katere te poteze vodijo. Stanja smo pregledali v padajočem vrstnem redu glede na vrednosti iz transpozicijske tabele, ki je nastala ob preiskovanju igralnega drevesa na prejšnji globini. Stanjem, ki še niso imela shranjenih ocen, smo pri tem pripisali nevtralno vrednost 0 in jih pregledali za stanji s pozitivnimi ocenami in pred tistimi z negativnimi ocenami. Ker v prvi iteraciji stara transpozicijska tabela še ni obstajala, potez takrat nismo sortirali.

Grafa 4.6a in 4.6b prikazujeta učinkovitost različnih stopenj optimiza-

[3]Vsaka poteza na ploščo doda točno en kamenček in ne odstrani nobenega, zato imajo vsa stanja na isti globini igralnega drevesa enako število kamenčkov. Stanja na globini $d + 1$ imajo en kamenček več od stanj na globini d.

cije pri preiskovanju dreves različnih globin. Merili smo čas izbire poteze v začetnem stanju igre, ko je igralna plošča še prazna. Opazimo lahko, da sta obe izboljšavi korenito zmanjšali iskalni čas pri globinah, večjih od 2. Rezultati meritev na začetni globini so približno enaki, saj program takrat zaradi neobstoja stare transpozicijske tabele še ne more koristiti prednosti optimizacij.



(a) Plošča velikosti 7×7.

(b) Plošča velikosti 11×11.

Slika 4.6: Učinkovitost transpozicijske tabele (TT) in sortiranja potez (SP) pri različnih globinah dreves na ploščah velikosti 7×7 in 11×11. Višina stolpca predstavlja razmerje med časom, ki ga je potreboval optimiziran program in časom, ki ga je za isto nalogo porabil program brez optimizacije. Številke nad stolpci so izmerjeni časi v sekundah.

4.5 Razviti igralci

Ustvarili smo 6 spodaj naštetih tipov igralcev z različnimi igralnimi strategijami.

Človek je igralec, ki ga usmerja človek preko internetnega brskalnika.

RAND naključno izbere eno izmed veljavnih potez.

MCTS(n) za izbiro potez uporablja le MCTS.

ABDT(n) uporablja minimaks z α-β in ocenjevalno funkcijo, pridobljeno z uporabo odločitvenega drevesa.

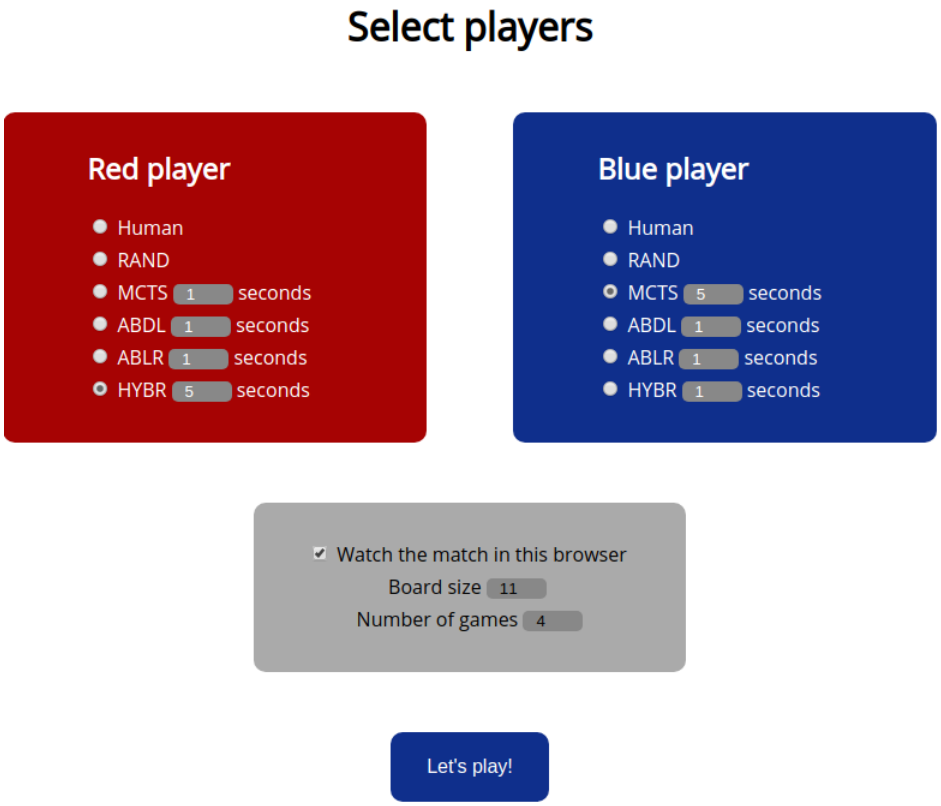
ABLR(n) je enak ABDT(n), le da uporablja ocenjevalno funkcijo, osnovano na linearni regresiji.

HYBR(n) je kombinacija ABLR(n) in MCTS(n). Pri izbiri prvih nekaj potez igra kot ABLR(n), v drugem delu pa kot MCTS(n). Meja, pri kateri se način igranja zamenja, je bila določena empirično.

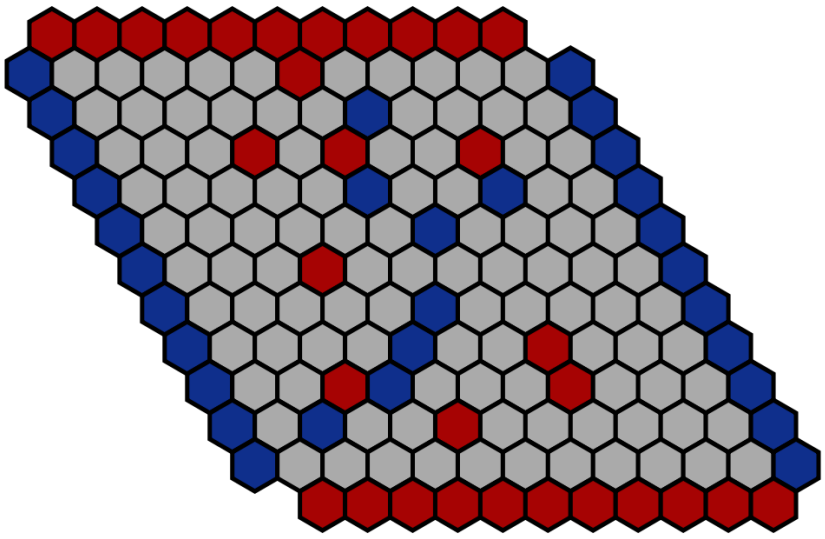
Pri zadnjih štirih igralcih smo dodali možnost nastavljanja časovne omejitve, v kateri mora igralec izbrati potezo: n predstavlja število sekund, ki jih je imel igralec na voljo za izbiro. Igralec HYBR ima v obeh načinih igre enako omejitev.

4.6 Grafični uporabniški vmesnik

Izdelali smo uporabniški vmesnik, ki je zelo enostaven, sestavljen iz le dveh delov: strani za nastavitev igre in strani z igralno ploščo. Oba dela sta prikazana na slikah 4.7 in 4.8. Vmesnik smo uporabljali za lažje sledenje poteku igre, igro človeka proti računalniku in osnovno primerjavo med igralci. Pravo testiranje smo izvedli neposredno na strežniku, tako da smo programu podali velikost plošče, seznam parov igralcev in časovno omejitev za vsakega igralca posebej.



Slika 4.7: Del grafičnega vmesnika, kjer je mogoče določiti lastnosti obeh igralcev, velikost igralne plošče in število odigranih iger.



Slika 4.8: Del grafičnega vmesnika, ki prikazuje potek igre.

Poglavje 5

Rezultati

Za preverjanje kakovosti igranja smo priredili manjši turnir med petimi nečloveškimi igralci. Igralcem smo pustili, da odigrajo več iger, pri čemer smo poskrbeli, da je vsak igralec v boju proti vsakemu drugemu igralcu

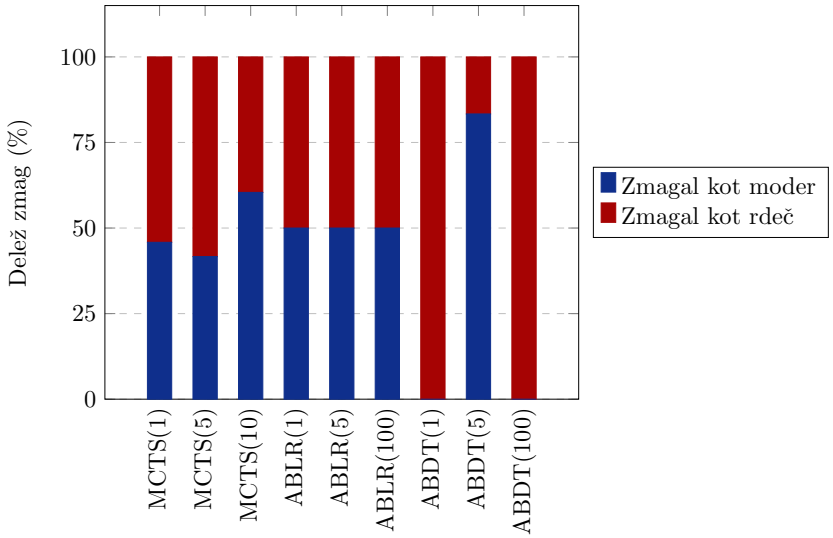
- polovico iger odigral z rdečimi kamenčki in polovico z modrimi,
- polovico iger odigral kot prvi igralec (tisti, ki igro začne) in polovico kot drugi,
- polovico iger, ki jih je odigral kot prvi igralec, igral z rdečimi kamenčki in polovico z modrimi.

V prvem podpoglavju predstavimo rezultate iger, ki so jih igralci dosegli v igrah proti samim sebi. Temu sledi podpoglavje o uspešnosti naprednejših igralcev proti igralcu RAND. Podpoglavje 5.3 prikaže rezultate iger med MCTS, ABDT in ABLR. V zadnjem delu primerjamo še MCTS in HYBR.

5.1 Igre proti samemu sebi

Preverjanje iger med istovrstnimi igralci se nam je zdelo smiselno le za igralce MCTS, ABDT in ADLR, saj igralec RAND izbira poteze povsem neodvisno od svoje barve, HYBR pa je le kombinacija ABLR in MCTS.

Vsem izbranim igralcem smo določili tri različne časovne omejitve: 1, 5 in 10 oz. 100 sekund. Igralcem, ki uporabljajo minimaks z α-β, smo pustili več časa (100 sekund), da bi ob izbiri poteze pravočasno končali pregledovanje čim globljega drevesa. Rezultate 48 iger, ki jih je vsak igralec odigral proti sebi, prikazuje graf 5.1.



Slika 5.1: Rezultati iger proti samemu sebi, pri čemer je vsak igralec odigral 48 iger. Modri del stolpca predstavlja delež iger, v katerih je zmagal modri igralec, rdeči del pa delež zmag rdečega igralca.

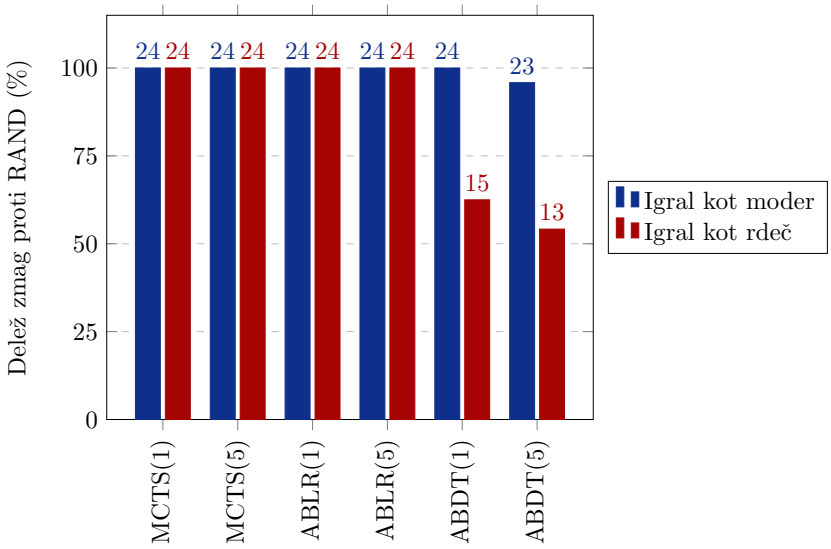
Razmerja zmag med modrim in rdečim igralcem pri MCTS in ABLR kažejo na to, da sta igralca obeh barv enakovredna. Pri ABLR smo opazili celo, da je vedno zmagal igralec, ki je igro začel, ne glede na svojo barvo. To se ujema z dokazano trditvijo o igri Hex, ki pravi, da lahko igralec, ki igro začne, vedno zmaga.

Rezultati ABDT so zelo zanimivi: pri časovni omejitvi 5 sekund je bil močnejši modri igralec, medtem ko je pri omejitvah 1 in 100 sekund vedno zmagal rdeči. Tega pojava si ne znamo točno pojasniti, sumimo pa, da je do razlike prihajalo v zadnjem delu igre. Opazili smo namreč, da so bile igre med dvema igralcema ABDT zelo dolge: število kamenčkov na plošči ob koncu igre je bilo v povprečju 90, večina ostalih iger pa se je končala že s

25 do 45 kamenčki. Predvidevamo, da je glavni razlog za dobljen rezultat vejitveni faktor, ki je ob veliki zasedenosti plošče dosti manjši kot na začetku igre. Dosežena globina preiskovanja se zato z višanjem časovne omejitve hitro povečuje, kar lahko močno vpliva na izbiro potez.

5.2 Igre proti RAND

Da bi preverili, ali naši igralci igrajo bolje, kot če bi samo naključno izbirali poteze, smo jih primerjali z igralcem RAND. Predvidevali smo, da bodo MCTS, ABDT in ABLR zmagovali že pri nižjih časovnih omejitvah, zato smo za omejitvi uporabili le 1 in 5 sekund. Med vsakim parom smo izpeljali 48 iger. Rezultate prikazuje graf 5.2.



Slika 5.2: Rezultati iger proti RAND. Vsak izmed izbranih igralcev je proti RAND odigral 24 iger kot modri in 24 kot rdeči igralec. Rdeči stolpci prikazujejo deleže/število zmag izbranih igralcev, ko so ti igrali z rdečimi kamenčki, modri stolpci pa deleže/število zmag izbranih igralcev, ko so uporabljali modre kamenčke.

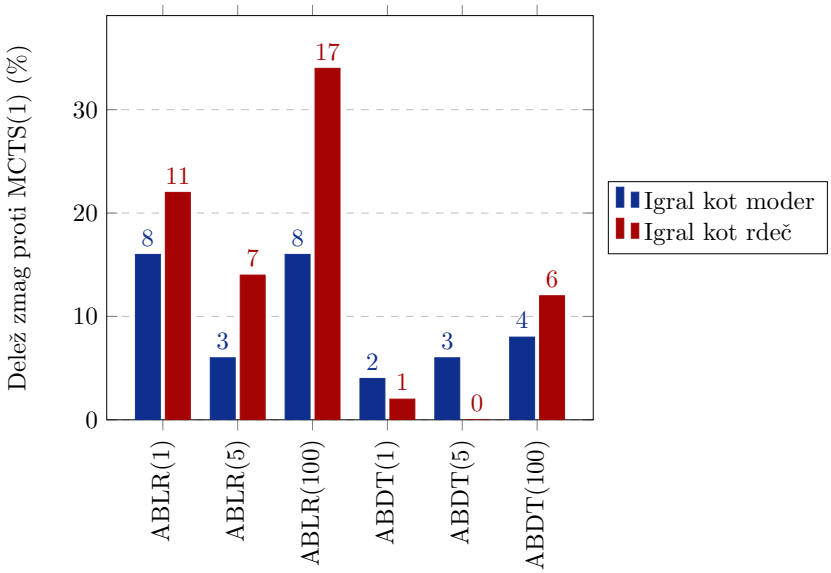
Po pričakovanjih sta MCTS in ABLR igrala dobro, zmagala sta v vseh igrah. Rezultati ABDT pa so ponovno presenetljivi: ko je igral kot modri

igralec, je izgubil samo eno igro, kot rdeči pa je zmagal le malo več kot v polovici iger. Tokrat je razlog bolj očiten. Zasledili smo, da ocenjevalna funkcija, ki jo uporablja ABDT, številnim stanjem pripiše isto vrednost. Pogosto se zgodi, da je več vrednosti β v vozliščih tik pod korenom drevesa ob zaključku preiskovanja z algoritmom minimaks enakih. Če se tudi najvišja izmed vseh vrednosti β na tem nivoju drevesa večkrat ponovi, program izbere prvo izmed potez s to oceno. Ker so možne poteze podane po vrstah igralne plošče od zgoraj navzdol, se večkrat izbere poteza, ki položi kamenček v eno izmed zgornjih vrstic plošče. Zaradi tega ima modri igralec, ki povezuje levo in desno stranico plošče pri igri proti naključnemu igralcu prednost pred rdečim. Modri namreč lahko zmaga samo z izbiro polj v zgornjem delu plošče, medtem ko rdeči potrebuje tudi kamenčke na dnu plošče.

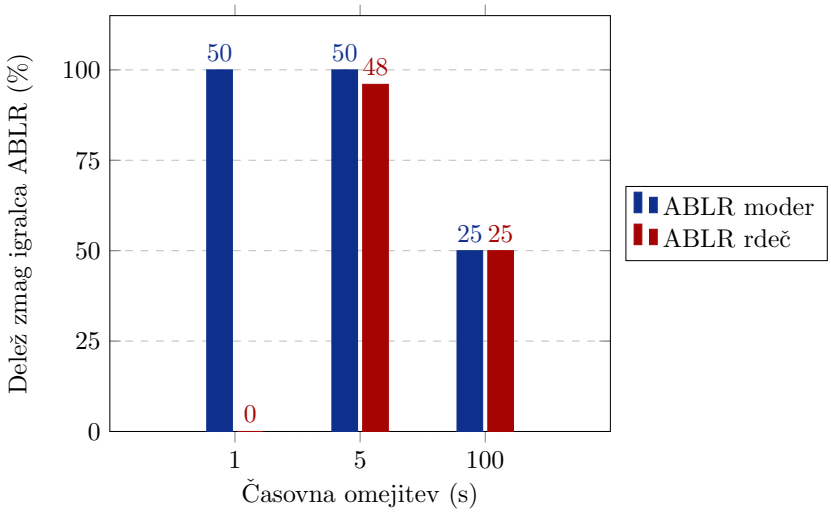
5.3 Igre med MCTS(1), ABDT in ABLR

Ker smo že pred začetkom testiranja ugotovili, da MCTS igra bolje od ABDT in ABLR, smo temu igralcu za izbiro poteze dovolili le eno sekundo, drugima dvema pa 1, 5 ali 100 sekund. Graf 5.3 kaže, da igralcema, ki uporabljata minimaks z α-β, veliko več časa ni pomagalo, da bi bila enakovredna MCTS, je pa to vseeno malo izboljšalo njuno igro.

Iz grafa 5.3 je razvidno tudi, da je ABLR proti MCTS dosegel boljše rezultate od ABDT. Zanimalo nas je, ali se bolje odreže tudi v neposrednem boju, zato smo ju primerjali še med seboj. Graf 5.4 prikazuje rezultate 100 odigranih iger pri vsaki časovni omejitvi (1, 5 in 100 sekund), pri čemer sta oba igralca znotraj ene igre imela enako omejitev. Razmerja zmag so na prvi pogled nenavadna, a se jih da enostavno pojasniti. V naši implementaciji algoritma minimaks z α-β ni nič prepuščeno naključju: veljavne poteze so vedno naštete v istem vrstnem redu, ocenjevalna funkcija za isto stanje vedno vrne isto vrednost, izmed najvišje ocenjenih potez je vedno izbrana prva ... Edina spremenljivka, ki lahko povzroči spremembo poteka igre, je procesor, na katerem se program izvaja, saj lahko njegova zasedenost vpliva



Slika 5.3: Rezultati iger igralcev, ki temeljijo na minimaksu, proti MCTS(1). Stolpca pri vsakem igralcu prikazujeta deleža/števili zmag, ki jih je ta igralec dosegel v 100 igrah (50 v vsaki barvi) proti MCTS(1). Barva stolpca predstavlja barvo kamenčkov, s katero je izbrani igralec prišel do zmage.



Slika 5.4: Rezultati iger ABLR proti ABDT. Modri stolpci prikazujejo delež in število iger, v katerih je ABLR z modrimi kamenčki premagal ABDT, rdeči pa delež in število iger, v katerih je ABLR zmagal kot rdeči igralec.

na število pravočasno končanih iteracij iterativnega poglabljanja, s tem pa tudi na izbiro poteze.

Dobljeni rezultati potrjujejo domnevo, da se igre, ki se enako začnejo, enako odvijajo in tudi enako končajo. Pri časovni omejitvi 1 sekunde je vedno zmagal modri igralec, pri omejitvi 5 sekund pa (skoraj) vedno ABLR. Pri najvišji omejitvi sta bila igralca enakovredna, analiza iger pa je pokazala, da je bil potek igre določen že z barvo in tipom prvega igralca. Med 100 igrami pri omejitvi 100 sekund so bile namreč le 4[1] unikatne, vsaka se je ponovila 25-krat.

Za boljšo primerjavo bi bilo smiselno preveriti še rezultate iger, pri katerih bi nekaj začetnih potez določili vnaprej, tako da bi se vsaka igra začela drugače. Ob tem bi morali paziti, da možnosti obeh igralcev za zmago ohranimo nespremenjene. Zaradi različnih začetkov bi bila vsaka igra drugačna, kar bi bolje prikazalo kakovost obeh igralcev.

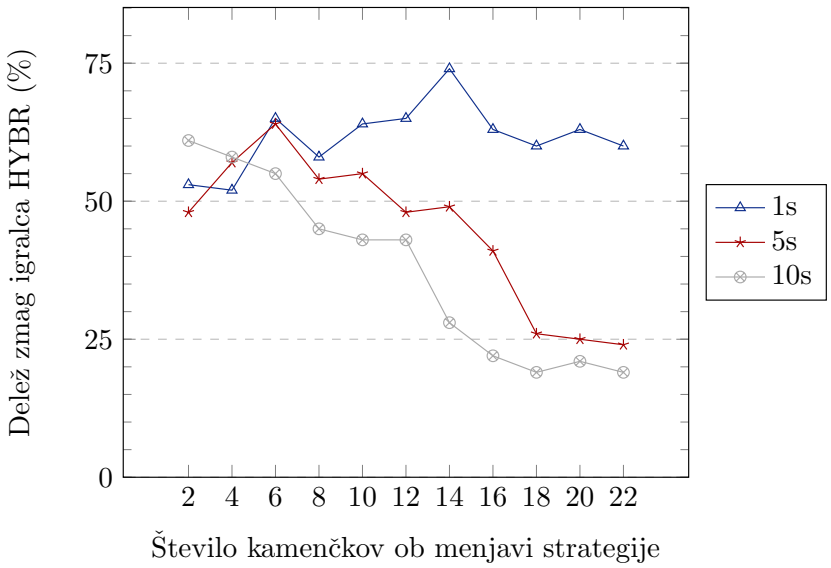
5.4 Igre med MCTS in HYBR

Ob opazovanju načina igranja MCTS in ABLR smo odkrili, da ABLR posebej dobro igra v začetnem delu igre, MCTS pa izbira dobre poteze takrat, ko je konec igre že blizu. To nas je privedlo na misel, da bi ustvarili igralca, ki bi kombiniral njune dobre lastnosti. Tako je nastal igralec HYBR, ki za začetne poteze uporablja minimaks z α-β, v drugem delu pa algoritem MCTS. Zanimalo nas je, ali je tak igralec lahko boljši od igralca MCTS.

Mejo, pri kateri pride do menjave strategije, smo določili s poskusi. Za možne meje (skupno število kamenčkov na igralni plošči) smo določili vsa soda števila med 2 in vključno 22. Za vsako smo med igralcema izpeljali 100 iger pri časovnih omejitvah 1, 5 in 100 sekund ter rezultate predstavili na grafu 5.5.

Sodeč po izidih je HYBR ob dobro izbrani meji menjave boljši od MCTS.

[1]Obstajajo 4 možne kombinacije za prvega igralca: rdeč ABDT, moder ABDT, rdeč ABLR in moder ABLR.



Slika 5.5: Delež zmag HYBR proti MCTS v odvisnosti od števila kamenčkov na plošči, pri katerem HYBR zamenja svoj način igranja. Različne črte predstavljajo rezultate pri različnih časovnih omejitvah (znotraj ene igre imata oba igralca enako omejitev).

Pri omejitvi 1 sekunde prevladuje vedno, najbolj pa, če spremeni način igranja po 14 kamenčkih. Pri omejiti 5 sekund se je za najboljšo izbiro izkazala meja 6, pri omejitvi 10 sekund pa 2 kamenčka. Pri slednjih dveh je moč opaziti, da z višanjem meje uspešnost hitro pada. Razlog za to je v tem, da zna MCTS dodatek časa izkoristiti bolje od ABLR. MCTS lahko v 10 sekundah izvede desetkrat več simulacij igre kot v eni sekundi, pri ABLR pa se ta časovna razlika dosti manj pozna, saj se v začetnem delu igre v devetih dodatnih sekundah zaključi kvečjemu ena iteracija algoritma več. Predvidevamo, da bi še višja omejitev izničila vso prednost igralca HYBR.

Poglavje 6

Zaključek

S tem diplomskim delom smo želeli preveriti, ali je mogoče hevristično funkcijo za uporabo v algoritmu minimaks ustvariti avtomatsko, brez človekovega vpliva na ločevanje med dobrimi in slabimi potezami ali stanji. Zanimalo nas je, v kolikšni meri lahko računalnik sam določi, katere lastnosti stanja v igri vplivajo na njegovo koristnost za posameznega igralca. Spodaj povzamemo svoja odkritja, diplomsko delo pa zaključimo s predlogi za nadaljnji razvoj.

6.1 Sklepne ugotovitve

Rezultati so pokazali, da se je naš sistem z linearno regresijo naučil sestaviti dovolj dobro ocenjevalno funkcijo, da njena uporaba v algoritmu minimaks z α-β vodi do solidnega igralca igre Hex. Hkrati pa ta funkcija ni dovolj točna, da bi omogočala dobro igro proti igralcem, ki temeljijo na simulacijah – v našem primeru MCTS.

Menimo, da glavni razlog za slabše rezultate proti MCTS tiči v kratkovidnosti naučene funkcije, ki z opazovanjem postavitve kamenčkov na plošči ne zna predvideti potencialnih virtualnih povezav. V kombinaciji z nizko globino pregledanega igralnega drevesa se je to pokazalo kot zelo slaba lastnost za boj z MCTS, ki uporablja simulacijo iger in odkrije najverjetnejši (ali vsaj zelo verjeten) potek igre.

Izkazalo se je, da lahko z naučeno funkcijo izboljšamo igralca MCTS. Ta na začetku igre potrebuje ogromno simuliranih iger, da lahko zanesljivo izbere dobro potezo. Nasprotno igralcema ABDT in ABLR za prvo oceno zadošča že pregled dveh nivojev igralnega drevesa, zato lahko služita kot otvoritvena knjižnica. V zaključnem delu igre pa ima prednost MCTS, ki hitreje odkrije pot do zmage ali poraza. S HYBR, kombinacijo ABLR in MCTS, nam je uspelo razviti igralca, ki združuje prednosti obeh in igra bolje od MCTS.

6.2 Možnosti nadaljnjega razvoja

Projekt odpira številne možnosti za nadaljevanje dela, spodaj so naštete le tiste, za katere menimo, da bi korenito vplivale na potek in kakovost igranja.

Prva možnost je upoštevanje pravila menjave, opisanega v poglavju 2.1, ki smo ga mi zaradi poenostavitve izpustili. Ta dodatek bi še dodatno razširil igralno drevo in upočasnil preiskovanje, po drugi strani pa bi s tem igralca postala bolj enakovredna.

Algoritmu MCTS bi bilo v fazi simulacije iz lista smiselno dodati nekaj domenske logike, na primer večjo verjetnost postavitve kamenčka na sredinska polja v začetnem delu igre. Upoštevala bi se lahko tudi kakšna izmed lastnosti igralca MoHex, ki je predstavljen v poglavju 2.3, recimo takojšnja obramba napadenega mostu. S tem bi bil potek igre bolj podoben realni igri, kar bi prispevalo k hitrejšemu približevanju ocen stanj teoretičnim vrednostim.

Fazo izbire vozlišč pri MCTS je mogoče izboljšati z ustreznejšo izbiro vrednosti konstante C, saj je bila naša izbrana brez posebne utemeljitve. Eden izmed načinov za določitev vrednosti bi bila medsebojna primerjava igralcev MCTS z različnimi vrednostmi te konstante.

Veliko možnosti izboljšav ponuja tudi del, kjer smo uporabili strojno učenje. Predvsem je priporočljivo delo na atributih, saj so obstoječi medsebojno zelo odvisni. Poleg tega bi lahko preizkusili še več različnih vrednosti parametrov pri gradnji odločitvenega drevesa in drugače postavljene meje

pri delitvi učnih podatkov pred linearno regresijo. Morda je vredno poskusiti tudi učenje povsem novega modela, na primer metode k najbližjih sosedov ali nevronskih mrež.

Literatura

[1] Vadim V. Anshelevich. The Game of Hex: An Automatic Theorem Proving Approach to Game Programming. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 189–194. AAAI Press, July 2000.

[2] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte Carlo Tree Search in Hex. In *IEEE Transactions on Computational Intelligence and AI in Games*, volume 2, pages 251–258. IEEE, December 2010.

[3] Assumptions of Linear Regression. Dosegljivo: `https://www.statisticssolutions.com/assumptions-of-linear-regression/`. [Dostopano 18. 12. 2018].

[4] Yngvi Björnsson. Minimax / Negamax / Alpha-Beta Pruning Algorithms. Prosojnice za pouk predmeta Artificial Intelligence na Reykjavik University.

[5] Shimon Even and Robert Endre Tarjan. A Combinatorial Problem Which Is Complete in Polynomial Space. *Journal of the ACM*, 23(4):710–719, October 1976.

[6] Martin Gardner. *Hexaflexagons and Other Mathematical Diversions: The First Scientific American Book of Puzzles and Games*. University Of Chicago Press, USA, September 1988.

[7] Sylvain Gelly and David Silver. Combining Online and Offline Knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning*, volume 227, pages 273–280. ACM, June 2007.

[8] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[9] ICGA Tournaments. Dosegljivo: `https://www.game-ai-forum.org/icga-tournaments/game.php?id=7`. [Dostopano 18. 5. 2018].

[10] Gabor Melis. Six. Dosegljivo: `http://six.retes.hu/`, 1999. [Dostopano 18. 5. 2018].

[11] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, Edouard Duchesnay, and Gilles Louppe. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, November 2011.

[12] Stefan Reisch. Hex ist PSPACE-vollständig. *Acta Informatica*, 15(2):167–191, June 1981.

[13] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[14] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[15] Michael Waskom, Olga Botvinnik, Drew O'Kane, Paul Hobson, Joel Ostblom, Saulius Lukauskas, David C Gemperline, Tom Augspurger, Yaroslav Halchenko, John B. Cole, Jordi Warmenhoven, Julian de Ruiter, Cameron Pye, Stephan Hoyer, Jake Vanderplas, Santi Villalba,

Gero Kunter, Eric Quintero, Pete Bachant, Marcel Martin, Kyle Meyer, Alistair Miles, Yoav Ram, Thomas Brunner, Tal Yarkoni, Mike Lee Williams, Constantine Evans, Clark Fitzgerald, Brian, and Adel Qalieh. mwaskom/seaborn: v0.9.0 (july 2018), July 2018.

Dodatek A

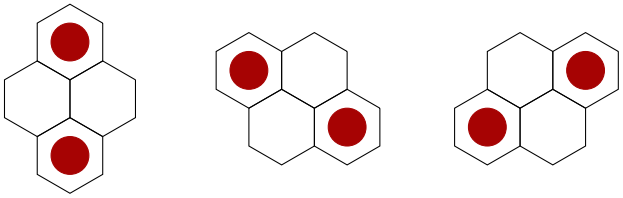
Iskani vzorci

Večino atributov, ki smo jih izbrali za opisovanje igralnih plošč, predstavljajo vzorci kamenčkov. Spodaj so našteti vsi iskani vzorci in njihove orientacije.

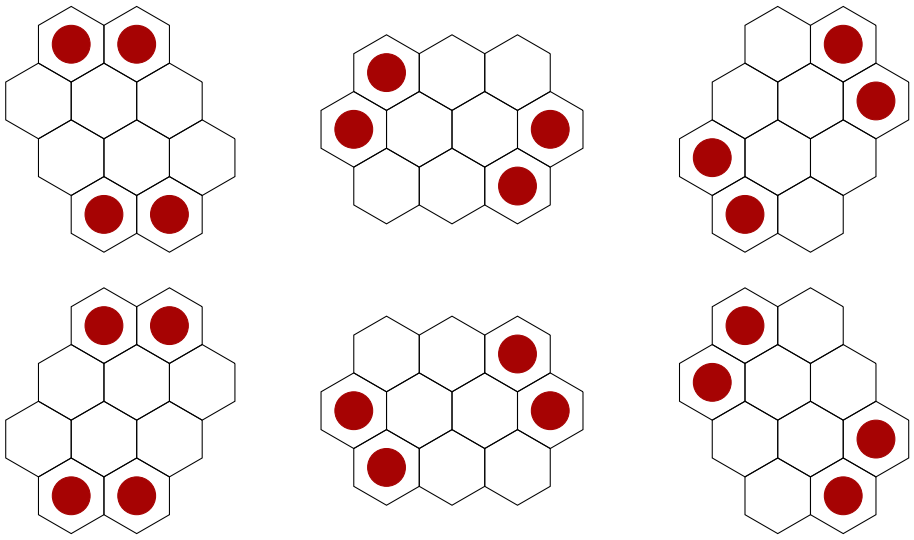
Čeprav nekatere orientacije izgledajo enako, smo jih ločili, saj niso enakovredne. Vzemimo za primer vzorce red_p1, red_p2 in red_p3, prikazane na sliki A.2. Prvi vzorec prekrije 3 vrste (in 2 stolpca) igralne plošče, zato je bolj koristen za igralca, ki povezuje zgornjo in spodnjo stranico igralne plošče. Nasprotno je tretji vzorec koristnejši za igralca, ki povezuje levi in desni stolpec, saj prekrije 3 stolpce (in le 2 vrstici) na plošči. Srednji vzorec ima za oba igralca enako vrednost.

Narisani so vzorci, ki smo jih iskali za rdečega igralca. Vzorci za modrega so bili enaki, le barvi kamenčkov sta bili zamenjani.

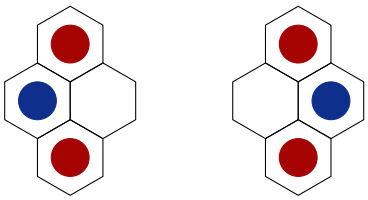
Slika A.1: Vzorec red_p0.



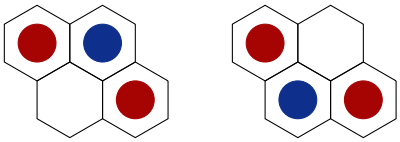
Slika A.2: Vzorci red_p1, red_p2 in red_p3.



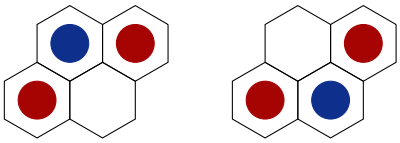
Slika A.3: Vse možne orientacije vzorca red_p4.



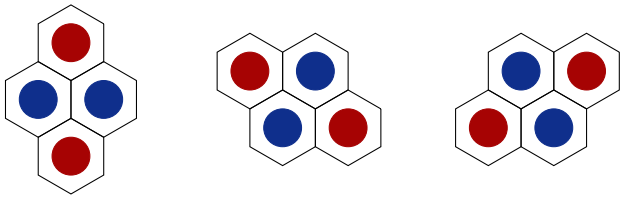
Slika A.4: Obe možni orientaciji vzorca red_p5.



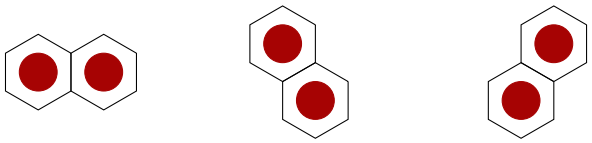
Slika A.5: Obe možni orientaciji vzorca red_p6.



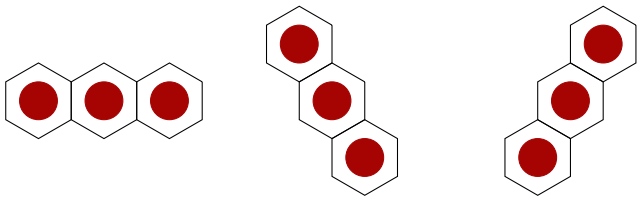
Slika A.6: Obe možni orientaciji vzorca red_p7.



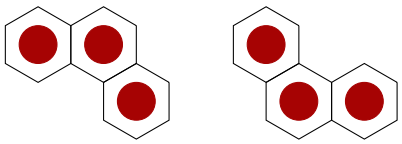
Slika A.7: Vzorci red_p8, red_p9 in red_p10.



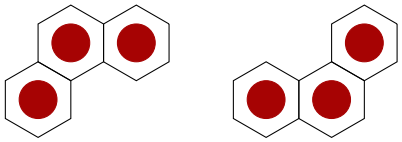
Slika A.8: Vzorci red_p11, red_p12 in red_p13.



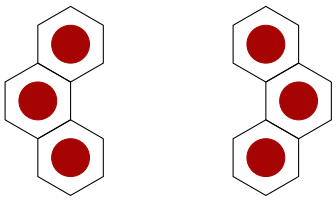
Slika A.9: Vzorci red_p14, red_p15 in red_p16.



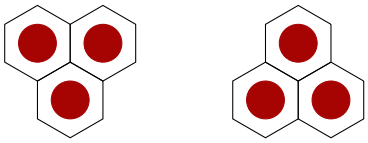
Slika A.10: Obe možni orientaciji vzorca red_p17.



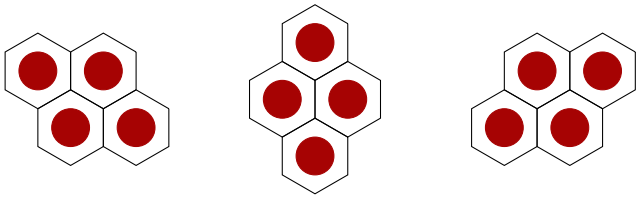
Slika A.11: Obe možni orientaciji vzorca red_p18.



Slika A.12: Obe možni orientaciji vzorca red_p19.



Slika A.13: Obe možni orientaciji vzorca red_p20.



Slika A.14: Vzorci red_p21, red_p22 in red_p23.

Dodatek B

Izpisana koda

V tem dodatku predstavimo kodo, ki je nastala v okviru strojnega učenja. Funkcija, ki je bila izpisana na podlagi odločitvenega drevesa, ima 236.590 vrstic, njen začetek prikazuje slika B.1. Na sliki B.2 je nekaj začetnih vrstic funkcije, osnovane na linearni regresiji, ki je v celoti dolga 1.378 vrstic.

Za lažjo predstavo pojasnimo uporabljena imena atributov:

lp Zadnji igralec na potezi.

num_stones Število kamenčkov na plošči.

stdc_r, stdc_b Vsota razdalj vseh rdečih (r) ali modrih (b) kamenčkov do sredine plošče.

rec_r, rec_b Število praznih polj plošče, ki se neposredno ali preko virtualne povezave dotikajo kamenčkov rdeče ali modre barve.

occ_red_rows, occ_blue_rows, occ_red_cols, occ_blue_cols Število vrstic (rows) ali stolpcev (cols) z vsaj enim rdečim (red) ali modrim (blue) kamenčkom.

red_pn, blue_pn Število pojavitev vzorca pn na plošči za rdečega ali modrega igralca.

```
func getEstimatedValueDT(s *Sample) float64 {
  if s.red_p9 <= 1 {
    if s.red_p17 <= 5 {
      if s.blue_p14 <= 1 {
        if s.red_p15 <= 0 {
          if s.sdtc_r <= 102 {
            if s.red_p12 <= 1 {
              if s.blue_p11 <= 0 {
                if s.rec_r <= 11 {
                  if s.rec_b <= 11 {
                    if s.lp <= 0 {
                      if s.rec_b <= 6 {
                        if s.rec_r <= 10 {
                          if s.rec_r <= 6 {
                            if s.occ_blue_rows <= 0 {
                              if s.sdtc_r <= 4 {
                                return -0.067017
                              }
                              if s.sdtc_r <= 9 {
                                return -0.0115455
                              }
                              return -0.023771
                            }
                            if s.rec_r <= 3 {
                              return -0.05448
                            }
                            if s.rec_r <= 5 {
                              if s.rec_b <= 5 {
                                if s.rec_b <= 4 {
                                  return -0.039216
                                }
                                return -0.036932666666666676
                              }
                              return -0.012683
                            }
                            ⋮
```

Slika B.1: Začetni del izpisane Go kode, generirane na podlagi odločitvenega drevesa.

```
func getEstimatedValueLR(s *Sample) float64 {
  if s.lp == 0 {
    switch {
    case s.num_stones <= 3:
      return (-557706280441.586)*float64(s.num_stones) +
      (-0.01170660244615059)*float64(s.sdtc_r) +
      (0.011634700062021035)*float64(s.sdtc_b) +
      (0.0016207244391313973)*float64(s.rec_r) +
      (-0.0015340438837465125)*float64(s.rec_b) +
      (0.0070752280775133545)*float64(s.occ_red_rows) +
      (-0.009298851027476017)*float64(s.occ_red_cols) +
      (130615426668.8434)*float64(s.occ_blue_rows) +
      (234991874288.91315)*float64(s.occ_blue_cols) +
      (557706280441.7081)*float64(s.red_p0) +
      (0.012282978884361959)*float64(s.red_p1) +
      (0.0072660659102118455)*float64(s.red_p2) +
      (0.024532978506079425)*float64(s.red_p3) +
      (0.002758680670824912)*float64(s.red_p5) +
      (-0.016999419953038712)*float64(s.red_p11) +
      (0.01777746983446039)*float64(s.red_p12) +
      (0.007560577900285134)*float64(s.red_p13) +
      (192098979483.7114)*float64(s.blue_p0)
    case s.num_stones <= 5:
      return (2501473385675.4736)*float64(s.num_stones) +
      ⋮
```

Slika B.2: Začetni del izpisane Go kode, generirane na podlagi linearne regresije.